Programmer's Guide to the Java 2D™ API

Enhanced Graphics and Imaging for Java

Java™ 2 SDK, Standard Edition, 1.3 Version

July 15, 1999

A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Contents

Preface

This guide describes the features provided by the Java 2D™ API and illustrates how you can use the Java 2D API classes to enhance your applications. For additional information about the Java 2D APIs, see:

- The Java Tutorial, 2nd Volume. Available online at:
  `http://java.sun.com/docs/books/tutorial/2d/index.html`
- The 2D Text Tutorial. Available online from the Java Developer Connection:
  `http://developer.java.sun.com/developer/onlineTraining/Graphics/2DText/`
- The Java 2D Sample Programs. Available online at:
  `http://java.sun.com/products/java-media/2D/samples/index.html`
- The Java 2D Demo. Available from the Java 2D website:
  `http://java.sun.com/products/java-media/2D/index.html`

This information in this guide is organized into seven chapters:

Overview —introduces the packages and key classes in the Java 2D API.

Rendering with Graphics2D—describes the Java 2D API classes in the `java.awt` package and how to set up the `Graphics2D` rendering context.

Geometries—describes the Java 2D API classes in the `java.awt.geom` package and how to define and manipulate 2D shapes and areas.

Fonts and Text Layout—describes the Java 2D API classes in the `java.awt.font` package, how to specify and retrieve font information, and how to display and manipulate international text using the Java 2D text layout APIs directly.

Imaging—describes the Java 2D API classes in the `java.awt.image`, `java.awt.image.codec`, and `java.awt.image.renderable` packages and how to display and manipulate images and offscreen buffers.

Color—describes the Java 2D API classes in the `java.awt.color` package and color management.

Printing—describes the Java 2D API classes in the `java.awt.print` package and the Java 2D API printing model.

1

Java 2D™ API Overview

The Java 2D™ API enhances the graphics, text, and imaging capabilities of the Abstract Windowing Toolkit (AWT), enabling the development of richer user interfaces and new types of Java™ applications.

Along with these richer graphics, font, and image APIs, the Java 2D API supports enhanced color definition and composition, hit detection on arbitrary geometric shapes and text, and a uniform rendering model for printers and display devices.

The Java 2D API also enables the creation of advanced graphics libraries, such as CAD-CAM libraries and graphics or imaging special effects libraries, as well as the creation of image and graphic file read/write filters.

When used in conjunction with the Java Media Framework and other Java Media APIs, the Java 2D APIs can be used to create and display animations and other multimedia presentations. The Java Animation and Java Media Framework APIs rely on the Java 2D API for rendering support.

1.1 Enhanced Graphics, Text, and Imaging

Early versions of the AWT provided a simple rendering package suitable for rendering common HTML pages, but not full-featured enough for complex graphics, text, or imaging. As a simplified rendering package, the early AWT embodied specific cases of more general rendering concepts. The Java 2D™ API provides a more flexible, full-featured rendering package by expanding the AWT to support more general graphics and rendering operations.

For example, through the `Graphics` class you can draw rectangles, ovals, and polygons. `Graphics2D` enhances the concept of geometric rendering by providing a mechanism for rendering virtually any geometric shape. Similarly, with the Java 2D API you can draw styled lines of any width and fill geometric shapes with virtually any texture.

Geometric shapes are provided through implementations of the `Shape` interface, for example `Rectangle2D` and `Ellipse2D`. Curves and arcs are also specific implementations of `Shape`.

Fill and pen styles are provided through implementations of the `Paint` and `Stroke` interfaces, for example `BasicStroke`, `GradientPaint`, `TexturePaint`, and `Color`.

`AffineTransform` defines linear transformations of 2D coordinates, including scale, translate, rotate, and shear.

Clip regions are defined by the same implementations of the `Shape` interface that are used to define general clipping regions, for example `Rectangle2D` and `GeneralPath`.

Color composition is provided by implementations of the `Composite` interface, for example `AlphaComposite`.

A `Font` is defined by collections of `Glyphs`, which are in turn defined by individual `Shapes`.

1.2 Rendering Model

The basic graphics rendering model has not changed with the addition of the Java 2D™ APIs. To render a graphic, you set up the graphics context and invoke a rendering method on the `Graphics` object.

The Java 2D API class `Graphics2D` extends `Graphics` to support more graphics attributes and provide new rendering methods. Setting up a `Graphics2D` context is described in "Rendering with Graphics2D" on page 15.

The Java 2D API automatically compensates for differences in rendering devices and provides a uniform rendering model across different types of devices. At the application level, the rendering process is the same whether the target rendering device is a display or a printer.

1.2.1 Coordinate Systems

The Java 2D API maintains two coordinate systems:

- *User space* is a device-independent, logical coordinate system. Applications use this coordinate system exclusively; all geometries passed into Java 2D rendering routines are specified in user space.

- *Device spac*e is a device-dependent coordinate system that varies according to the target rendering device.

The Java 2D system automatically performs the necessary conversions between user space and the device space of the target rendering device. Although the coordinate system for a monitor is very different from the coordinate system for a printer, these differences are invisible to applications.

1.2.1.1 User Space

As shown in Figure 1-1, the user space origin is located in the upper-left corner of the space, with *x* values increasing to the right and *y* values increasing downward.

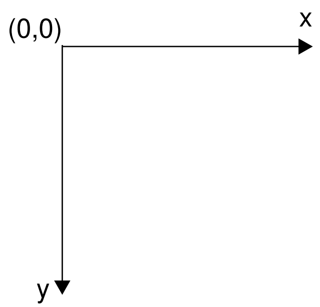


Figure 1-1 User Space Coordinate System

User space represents a uniform abstraction of all possible device coordinate systems. The device space for a particular device might have the same origin and direction as user space, or it might be different. Regardless, user space coordinates are automatically transformed into the appropriate device space when a graphic object is rendered. Often, the underlying platform device drivers are used to perform this conversion.

1.2.1.2 Device Space

The Java 2D API defines three levels of configuration information that are maintained to support the conversion from user space to device space. This information is encapsulated by three classes:

- `GraphicsEnviron`ment
- `GraphicsDevice`
- `GraphicsConfiguration`

The GraphicsEnvironment describes the collection of rendering devices visible to a Java application on a particular platform. Rendering devices include screens, printers, and image buffers. The GraphicsEnvironment also includes a list of all of the available fonts on the platform.

A GraphicsDevice describes an application-visible rendering device, such as a screen or printer. Each possible configuration of the device is represented by a GraphicsConfiguration. For example, an SVGA display device can operate in several modes: 640x480x16 colors, 640x480x256 colors, and 800x600x256 colors. The SVGA screen is represented by a GraphicsDevice object and each of the modes is represented by a GraphicsConfiguration object.

A GraphicsEnvironment can contain one or more GraphicsDevices; in turn, each GraphicsDevice can have one or more GraphicsConfigurations.

Between them, the GraphicsEnvironment, GraphicsDevice, and GraphicsConfiguration represent all of the information necessary for locating a rendering device or font on the Java platform and for converting coordinates from user space to device space. An application can access this information, but does not need to perform any transformations between user space and device space.

1.2.2 Transforms

The Java 2D API has a unified coordinate transformation model. All coordinate transformations, including transformations from user to device space, are represented by AffineTransform objects. AffineTransform defines the rules for manipulating coordinates using matrices.

You can add an AffineTransform to the graphics context to rotate, scale, translate, or shear a geometric shape, text, or image when it is rendered. The added transform is applied to any graphic object rendered in that context. The transform is performed when user space coordinates are converted to device space coordinates.

1.2.3 Fonts

A string is commonly thought of in terms of the characters that comprise the string. When a string is drawn, its appearance is determined by the font that is selected. However, the shapes that the font uses to display the string don't always correspond to individual characters. For example, in professional publishing, certain combinations of two or more characters are often replaced by a single shape called a *ligature*.

The shapes that a font uses to represent the characters in the string are called *glyphs*. A font might represent a character such as a lowercase *a* acute using multiple glyphs, or represent certain character combinations such as the *fi* in final with a single glyph. In the Java 2D API, a glyph is simply a `Shape` that can be manipulated and rendered in the same way as any other `Shape`.

A *font* can be thought of as a collection of glyphs. A single font might have many versions, such as heavy, medium, oblique, gothic, and regular. These different versions are called *faces*. All of the faces in a font have a similar typographic design and can be recognized as members of the same *family*. In other words, a collection of glyphs with a particular style forms a font face, a collection of font faces forms a font family, and a collection of font families forms the set of fonts available within a particular `GraphicsEnvironment`.

In the Java 2D API, fonts are specified by a name that describes a particular font face—for example, Helvetica Bold. This is different from the JDK 1.1 software, in which fonts are described by logical names that map onto different font faces depending on which font faces are available on a particular platform. For backward compatibility, the Java 2D API supports the specification of fonts by logical name as well as by font face name.

Using the Java 2D API, you can compose and render strings that contain multiple fonts of different families, faces, sizes, and even languages. The appearance of the text is kept logically separate from the layout of the text. `Font` objects are used to describe the appearance, and the layout information is stored in `TextLayout` and `TextAttributeSet` objects. Keeping the font and layout information separate makes it easier to use the same fonts in different layout configurations.

1.2.4 Images

Images are collections of pixels organized spatially. A *pixel* defines the appearance of an image at a single display location. A two-dimensional array of pixels is called a *raster*.

The pixel's appearance can be defined directly or as an index into a color table for the image.

In images that contain many colors (more than 256), the pixels usually directly represent the color, alpha, and other display characteristics for each screen location. Such images tend to be much larger than indexed-color images, but they look more realistic.

In an indexed-color image, the colors in the image are limited to the colors specified in the color table, often resulting in fewer colors that can be used in the image. However, an index typically requires less storage space than a color value, so images stored as a set of indexed colors are usually smaller. This pixel format is popular for images that contain only 16 or 256 colors.

Images in the Java 2D API have two primary components:

- The raw image data (the pixels)
- The information necessary for interpreting the pixels

The rules for interpreting the pixel are encapsulated by a `ColorModel` object—for example, whether the values should be interpreted as direct or indexed colors. For a pixel to be displayed, it must be paired with a color model.

A *band* is one component of the color space for an image. For example, the Red, Green, and Blue components are the bands in an RGB image. A pixel in a direct color model image can be thought of as a collection of band values for a single screen location.

The `java.awt.image` package contains several `ColorModel` implementations, including those for packed and component pixel representations.

A `ColorSpace` object encapsulates the rules that govern how a set of numeric measurements corresponds to a particular color. The `ColorSpace` implementations in the `java.awt.color` represent the most popular color spaces, including RGB and gray scale. Note that a color space is *not* a collection of colors—it defines the rules for how to interpret individual color values.

Separating the color space from the color model provides greater flexibility in how colors are represented and converted from one color representation to another.

1.2.5 Fills and Strokes

With the Java 2D API, you can render `Shapes` using different pen styles and fill patterns. Because text is ultimately represented by a set of glyphs, text strings can also be stroked and filled.

Pen styles are defined by objects that implement the `Stroke` interface. Strokes enable you to specify different widths and dashing patterns for lines and curves.

Fill patterns are defined by objects that implement the `Paint` interface. The `Color` class, which was available in earlier versions of the AWT, is a simple type of `Paint` object used to define solid-color fills. The Java 2D API provides two additional `Paint` implementations, `TexturePaint` and `GradientPaint`. `Texture-`

`Paint` defines a fill pattern using a simple image fragment that is repeated uniformly. `GradientPaint` defines a fill pattern as a gradient between two colors.

In Java 2D, rendering a shape's outline and filling the shape with a pattern are two separate operations:

- Using one of the `draw` methods renders the shape's contour or outline using the pen style specified by the `Stroke` attribute and the fill pattern specified by the `Paint` attribute.
- Using the `fill` method fills the interior of the shape with the pattern specified by the `Paint` attribute.

When a text string is rendered, the current `Paint` attribute is applied to the glyphs that form the string. Note, however, that `drawString` actually fills the glyphs that are rendered. To stroke the outlines of the glyphs in a text string, you need to get the outlines and render them as shapes using the `draw` method.

1.2.6 Composites

When you render an object that overlaps an existing object, you need to determine how to combine the colors of the new object with the colors that already occupy the area where you are going to draw. The Java 2D API encapsulates rules for how to combine colors in a `Composite` object.

Primitive rendering systems provide only basic Boolean operators for combining colors. For example, a Boolean compositing rule might allow the source and destination color values to be ANDed, ORed, or XORed. There are several problems with this approach

- It's not "human friendly"—it's difficult to think in terms of what the resulting color will be if red and blue are ANDed, not added.
- Boolean composition does not support the accurate composition of colors in different color spaces.
- Straight Boolean composition doesn't take into account the color models of the colors. For example, in an indexed color model, the result of a Boolean operation on two pixel values in an image is the composite of two indices, not two colors.

The Java 2D API avoids these pitfalls by implementing alpha-blending[1] rules that take color model information into account when compositing colors. An `AlphaComposite` object includes the color model of both the source and destination colors.

1.3 Backward Compatibility and Platform Independence

The Java 2D™ API maintains backward compatibility with JDK 1.1 software. It is also architected so that applications can maintain platform-independence.

1.3.1 Backward Compatibility

To ensure backward compatibility, the functionality of existing JDK graphics and imaging classes and interfaces was maintained. Existing features were not removed and no package designations were changed for existing classes. The Java 2D API enhances the functionality of the AWT by implementing new methods in existing classes, extending existing classes, and adding new classes and interfaces that don't affect the legacy APIs.

For example, much of the Java 2D API functionality is delivered through an expanded graphics context, `Graphics2D`. To provide this extended graphics context while maintaining backward compatibility, `Graphics2D` extends the `Graphics` class from the JDK 1.1 release.

The usage model of the graphics context remains unchanged. The AWT passes a graphics context to an AWT `Component` through the following methods:

- `paint`
- `paintAll`
- `update`
- `print`
- `printAll`
- `getGraphics`

A JDK 1.1 applet interprets the graphics context that's passed in as an instance of `Graphics`. To gain access to the new features implemented in `Graphics2D`, a Java 2D API–compatible applet casts the graphics context to a `Graphics2D` object:

```
public void Paint (Graphics g) {

    Graphics2D g2 = (Graphics2D) g;
```

1. For detailed information about alpha blending, see Section 17.6 of *Computer Graphics: Principles and Practice*. 2nd ed. J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes. Addison-Wesley, 1990.

```
        ...
        ...
        g2.setTransform (t);
    }
```

In some cases, rather than extending a legacy class, the Java 2D API generalizes it. Two techniques were used to generalize legacy classes:

- One or more parent classes were inserted in the hierarchy, and the legacy class was updated to extend the new parent classes. This technique is used to add general implemented methods and instance data to the legacy class.
- One or more interface implementations were added to the legacy class. This technique is used to add general abstract methods to the legacy class.

For example, the Java 2D API generalizes the AWT `Rectangle` class using both of these techniques. The hierarchy for rectangle now looks like:

```
java.lang.object
|
+-------java.awt.geom.RectangularShape
             |
             +---------java.awt.geom.Rectangle2D
                                |
                                +-------java.awt.Rectangle
```

In the JDK 1.1 software, `Rectangle` simply extended `Object`. It now extends the new `Rectangle2D` class and implements both `Shape` and `Serializable`. Two parent classes were added to the `Rectangle` hierarchy: `RectangularShape` and `Rectangle2D`. Applets written for JDK 1.1 software are unaware of the new parent classes and interface implementations, but are unaffected because `Rectangle` still contains the methods and members that were present in earlier versions.

The Java 2D API adds several new classes and interfaces that are “orthogonal” to the legacy API. These additions do not extend or generalize existing classes—they are entirely new and distinct. These new classes and interfaces embody concepts that had no explicit representation in the legacy API.

For example, the Java 2D API implements several new `Shape` classes, including `Arc2D`, `CubicCurve2D`, and `QuadCurve2D`. Although early versions of the AWT could render arcs using the `drawArc` and `fillArc` methods, there was no general curve abstraction and no discrete classes that embodied arcs. These discrete classes could be added to the Java 2D API without disrupting legacy applets because `drawArc` and `fillArc` are still supported through the `Graphics` class.

1.3.2 Platform Independence

To enable the development of platform-independent applications, the Java 2D API makes no assumptions about the resolution, color space, or color model of the target rendering device. Nor does the Java 2D API assume any particular image file format.

Truly platform-independent fonts are possible only when the fonts are built-in (provided as part of the JDK software), or when they are mathematically or programmatically generated. The Java 2D API does not currently support built-in or mathematically generated fonts, but it does enable the programmatic definition of entire fonts through their glyph set. Each glyph can in turn be defined by a `Shape` that consists of line segments and curves. Many fonts of particular styles and sizes can be derived from a single glyph set.

1.4 The Java 2D™ API Packages

The Java 2D API classes are organized into the following packages:

- `java.awt`
- `java.awt.geom`
- `java.awt.font`
- `java.awt.color`
- `java.awt.image`
- `java.awt.image.renderable`
- `java.awt.print`

Package `java.awt` contains those Java 2D API classes and interfaces that are general in nature or that enhance legacy classes. (Obviously, not all of the classes in `java.awt` are Java 2D classes.)

`AlphaComposite`	`BasicStroke`	`Color`
`Composite`	`CompositeContext`	`Font`
`GradientPaint`	`Graphics2D`	`GraphicsConfiguration`
`GraphicsDevice`	`GraphicsEnvironment`	`Paint`
`PaintContext`	`Rectangle`	`Shape`
`Stroke`	`TexturePaint`	`Transparency`

Package `java.awt.geom` contains classes and interfaces related to the definition of geometric primitives:

AffineTransform	Arc2D	Arc2D.Double
Arc2D.Float	Area	CubicCurve2D
CubicCurve2D.Double	CubicCurve2D.Float	Dimension2D
Ellipse2D	Ellipse2D.Double	Ellipse2D.Float
FlatteningPathIterator	GeneralPath	Line2D
Line2D.Double	Line2D.Float	PathIterator
Point2D	Point2D.Double	Point2D.Float
QuadCurve2D	QuadCurve2D.Double	QuadCurve2D.Float
Rectangle2D	Rectangle2D.Double	Rectangle2D.Float
RectangularShape	RoundRectangle2D	RoundRectangle2D.Double
RoundRectangle2D.Float		

Many of the geometric primitives have corresponding `.Float` and `.Double` implementations. This was done to enable both floating single- and double-precision implementations. Double-precision implementations provide greater rendering precision at the expense of performance on some platforms.

Package `java.awt.font` contains classes and interfaces used for text layout and the definition of fonts:

FontRenderContext	GlyphJustificationInfo	GlyphMetrics
GlyphVector	GraphicAttribute	ImageGraphicAttribute
LineBreakMeasurer	LineMetrics	MultipleMaster
OpenType	ShapeGrapicAttribute	TextAttribute
TextHitInfo	TextLayout	TransformAttribute

Package`java.awt.color` contains classes and interfaces for the definition of color spaces and color profiles:

ColorSpace	ICC_ColorSpace	ICC_Profile
ICC_ProfileGray	ICC_ProfileRGB	

The `java.awt.image` and `java.awt.image.renderable` packages contain classes and interfaces for the definition and rendering of images:

AffineTransformOp	BandCombineOp	BandedSampleModel
BufferedImage	BufferedImageFilter	BufferedImageOp
ByteLookupTable	ColorConvertOp	ColorModel
ComponentColorModel	ComponentSampleModel	ConvolveOp
ContextualRenderedImageFactory		DataBuffer
DataBufferByte	DataBufferInt	DataBufferShort
DataBufferUShort	DirectColorModel	IndexColorModel
Kernel	LookupOp	LookupTable
MultiPixelPackedSampleModel	PackedColorModel	ParameterBlock
PixelInterleavedSampleModel	Raster	RasterOp
RenderableImage	RenderableImageOp	RenderableImageProducer
RenderContext	RenderedImageFactory	RenderedImage
RescaleOp	SampleModel	ShortLookupTable
SinglePixelPackedSampleModel		TileObserver
WritableRaster	WritableRenderedImage	

Package `java.awt.image` was present in earlier versions of the AWT. The Java 2D API enhances the following legacy AWT image classes:

- `ColorModel`
- `DirectColorModel`
- `IndexColorModel`

These color model classes remain in the `java.awt.image` package for backward compatibility. To maintain consistency, the new color model classes are also located in the `java.awt.image` package.

Package `java.awt.print` contains classes and interfaces that enable printing of all Java 2D–based text, graphics, and images.

Book	Pageable	PageFormat
Paper	Printable	PrinterGraphics

PrinterJob

2

Rendering with Graphics2D

`Graphics2D` extends `java.awt.Graphics` to provide more sophisticated control over the presentation of shapes, text, and images. The Java 2D™ rendering process is controlled through the `Graphics2D` object and its state attributes.

The `Graphics2D` state attributes, such as line styles and transformations, are applied to graphic objects when they are rendered. The collection of state attributes associated with a `Graphics2D` is referred to as the `Graphics2D` *context*. To render text, shapes, or images, you set up the `Graphics2D` context and then call one of the `Graphics2D` rendering methods, such as `draw` or `fill`.

2.1 Interfaces and Classes

The following tables list the interfaces and classes used in conjunction with the `Graphics2D` context, including the classes that represent state attributes. Most of these classes are part of the `java.awt` package.

Interface	Description
`Composite`	Defines methods to compose a draw primitive with the underlying graphics area. Implemented by `AlphaComposite`.
`CompositeContext`	Defines the encapsulated and optimized environment for a composite operation. Used by programmers implementing custom compositing rules.
`Paint`	Extends: `Transparency` Defines colors for a `draw` or `fill` operation. Implemented by `Color`, `GradientPaint` and `TexturePaint`.
`PaintContext`	Defines the encapsulated and optimized environment for a paint operation. Used by programmers implementing custom paint operations.

Interface	Description
`Stroke`	Generates a `Shape` that encloses the outline of the `Shape` to be rendered. Implemented by `BasicStroke`.

Class	Description
`AffineTransform` (`java.awt.geom`)	Represents a 2D affine transform, which performs a linear mapping from 2D coordinates to other 2D coordinates.
`AlphaComposite`	Implements: `Composite` Implements basic alpha composite rules for shapes, text, and images.
`BasicStroke`	Implements: `Stroke` Defines the "pen style" to be applied to the outline of a `Shape`.
`Color`	Implements: `Paint` Defines a solid color fill for a `Shape`.
`GradientPaint`	Implements: `Paint` Defines a linear color gradient fill pattern for a `Shape`. This fill pattern changes from color C1 at point P1 to color C2 at point P2.
`Graphics2D`	Extends: `Graphics` Fundamental class for 2D rendering. Extends the original `java.awt.Graphics` class.
`TexturePaint`	Implements: `Paint` Defines a texture or pattern fill for a `Shape`. The texture or pattern is generated from a `BufferedImage`.

2.2 Rendering Concepts

To render a graphic object using the Java 2D™ API, you set up the `Graphics2D` context and pass the graphic object to one of the `Graphics2D` rendering methods.

You can modify the state attributes that form the `Graphics2D` context to:

- Vary the stroke width.
- Change how strokes are joined together.
- Set a clipping path to limit the area that is rendered.
- Translate, rotate, scale, or shear objects when they are rendered.
- Define colors and patterns to fill shapes with.
- Specify how multiple graphics objects should be composed.

`Graphics2D` defines several methods for adding and changing attributes in the graphics context. Most of these methods take an object that represents a particular attribute, such as a `Paint` or `Stroke` object.

The `Graphics2D` context holds *references* to these attribute objects: they are not cloned. If you alter an attribute object that is part of the `Graphics2D` context, you need to call the appropriate `set` method to notify the context. Modifying an attribute object during a rendering operation will cause unpredictable and possibly unstable behavior.

2.2.1 Rendering Process

When a graphic object is rendered, the geometry, image, and attribute information are combined to calculate which pixel values must be changed on the display.

The rendering process for a `Shape` can be broken down into four steps:

1. If the `Shape` is to be stroked, the `Stroke` attribute in the `Graphics2D` context is used to generate a new `Shape` that encompasses the stroked path.
2. The coordinates of the `Shape`'s path are transformed from user space into device space according to the transform attribute in the `Graphics2D` context.
3. The `Shape`'s path is clipped using the clip attribute in the `Graphics2D` context.
4. The remaining `Shape`, if any, is filled using the `Paint` and `Composite` attributes in the `Graphics2D` context.

Rendering text is similar to rendering a `Shape`, since the text is rendered as individual glyphs and each glyph is a `Shape`. The only difference is that the Java 2D API must determine what `Font` to apply to the text and get the appropriate glyphs from the `Font` before rendering.

Images are handled differently, transformations and clipping operations are performed on the image's bounding box. The color information is taken from the image itself and its alpha channel is used in conjunction with the current `Composite` attribute when the image pixels are composited onto the rendering surface.

2.2.2 Controlling Rendering Quality

The Java 2D API lets you indicate whether you want objects to be rendered as quickly as possible, or whether you prefer that the rendering quality be as high as possible. Your preferences are specified as hints through the `RenderingHints` attribute in the `Graphics2D` context. Not all platforms support modification of the rendering mode so specifying rendering hints does not guarantee that they will be used.

The `RenderingHints` class supports the following types of hints:

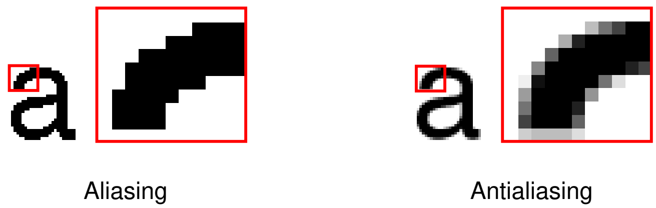
- Alpha interpolation—can be set to default, quality, or speed.
- Antialiasing—can be set to default, on, or off.
- Color Rendering–can be set to default, quality, or speed.
- Dithering—can be set to default, disable, or enable.
- Fractional Metrics—can be set to default, on, or off.
- Interpolation—can be set to nearest-neighbor, bilinear, or bicubic.
- Rendering—can be set to default, quality, or speed.
- Text antialiasing—can be set to default, on, or off.

To set or change the `RenderingHints` attribute in the `Graphics2D` context, you call `setRenderingHints`. When a hint is set to default, the platform rendering default is used is used.

Antialiasing

When graphics primitives are rendered on raster-graphics display devices, their edges can appear jagged because of *aliasing*. Arcs and diagonal lines take on a jagged appearance because they are approximated by turning on the pixels that are closest to the path of the line or curve. This is particularly noticeable on low-resolution devices, where the jagged edges appear in stark contrast to the smooth edges of horizontal or vertical lines.

Antialiasing is a technique used to render objects with smoother-appearing edges. Instead of simply turning on the pixel that is closest to the line or curve, the intensity of surrounding pixels is set in proportion to the amount of area covered by the geometry being rendered. This softens the edges and spreads the on-off transition over multiple pixels. However, antialiasing requires additional computing resources and can reduce rendering speed.



2.2.3 Stroke Attributes

Stroking a `Shape` such as a `GeneralPath` object is equivalent to running a logical pen along the segments of the `GeneralPath`. The `Graphics2D` `Stroke` attribute defines the characteristics of the mark drawn by the pen.

A `BasicStroke` object is used to define the stroke attributes for a `Graphics2D` context. `BasicStroke` defines characteristics such as the line width, endcap style, segment join-style, and the dashing pattern. To set or change the `Stroke` attribute in the `Graphics2D` context, you call `setStroke`.

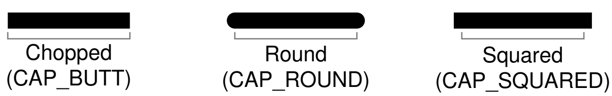


Figure 2-1 endcap styles supported by `BasicStroke`

Figure 2-2 Join styles supported by `BasicStroke`

For example, the first image in Figure 2-3 uses the miter join-style; the second image uses a round join-style, a round endcap style, and a dashing pattern.

Figure 2-3 Stroke Styles

The `Graphics2D` rendering methods that use the `Stroke` attribute are `draw`, `drawArc`, `drawLine`, `drawOval`, `drawPolygon`, `drawPolyline`, `drawRect`, and `drawRoundRect`.When one of these methods is called, the outline of the specified `Shape` is rendered. The `Stroke` attribute defines the line characteristics and the `Paint` attribute defines the color or pattern of the mark drawn by the pen.

For example, when `draw(myRectangle)` is called:

1. The `Stroke` is applied to the rectangle's outline.
2. The stroked outline is converted to a `Shape` object.
3. The `Paint` is applied to the pixels that lie within the contour of the outline `Shape`.

This process is illustrated in Figure 2-4:

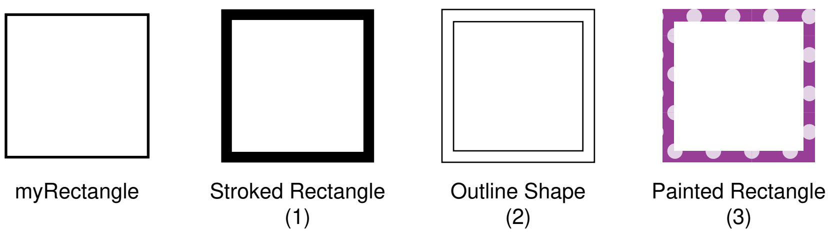


Figure 2-4 Stroking a Shape

2.2.4 Fill Attributes

The fill attribute in the `Graphics2D` context is represented by a `Paint` object. You add a `Paint` to the `Graphics2D` context by calling `setPaint`.

When a `Shape` or glyph is drawn (`Graphics2D.draw`, `Graphics2D.drawString`), the `Paint` is applied to all of the pixels that lie inside of the `Shape` that represents the object's stroked outline. When a `Shape` is filled (`Graphics2D.fill`), the `Paint` is applied to all of the pixels that lie within the `Shape`'s contour.

Simple solid color fills can be set with the `setColor` method. `Color` is the simplest implementation of the `Paint` interface.

To fill `Shapes` with more complex paint styles such as gradients and textures, you use the Java 2D `Paint` classes `GradientPaint` and `TexturePaint`. These classes eliminate the time-consuming task of creating complex fills using simple solid-color paints. Figure 2-5 illustrates two fills that could easily be defined by `GradientPaint` and `TexturePaint`.

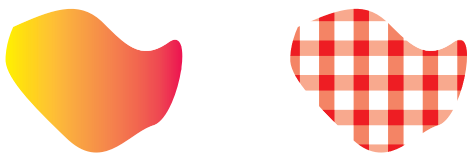


Figure 2-5 Complex Fill Styles

When `fill` is called to render a `Shape`, the system:

1. Determines what pixels comprise the `Shape`.
2. Gets the color of each pixel from the `Paint` object.
3. Converts the color to an appropriate pixel value for the output device.
4. Writes the pixel to that device.

Batch Processing

To streamline the processing of pixels, the Java 2D API processes them in batches. A batch can be either a contiguous set of pixels on a given scanline or a block of pixels. This batch processing is done in two steps:

1. The `Paint` object's `createContext` method is called to create a `PaintContext`. The `PaintContext` stores the contextual information about the current rendering operation and the information necessary to generate the colors. The `createContext` method is passed the bounding boxes of the graphics object being filled in user space and in device space, the `ColorModel` in which the colors should be generated, and the transform used to map user space into device space. The `ColorModel` is treated as a hint because not all `Paint` objects can support an arbitrary `ColorModel`. (For more information about `ColorModels`, see "Color" on page 89.")
2. The `getColorModel` method is called to get the `ColorModel` of the generated paint color from the `PaintContext`.

The `getRaster` method is then called repeatedly to get the `Raster` that contains the actual color data for each batch. This information is passed to the next stage in the rendering pipeline, which draws the generated color using the current `Composite` object.

2.2.5 Clipping Paths

A *clipping path* identifies the portion of a `Shape` or `Image` that needs to be rendered. When a clipping path is part of the `Graphics2D` context, only those parts of a `Shape` or `Image` that lie within the path are rendered.

To add a clipping path to the `Graphics2D` context, you call `setClip`. Any `Shape` can be used to define the clipping path.

To change the clipping path, you can either use `setClip` to specify a new path or call `clip` to change the clipping path to the intersection of the old clipping path and a new `Shape`.

2.2.6 Transformations

The `Graphics2D` context contains a transform that is used to transform objects from user space to device space during rendering. To perform additional transformations, such as rotation or scaling, you can add other transforms to the `Graphics2D` context. These additional transforms become part of the pipeline of transformations applied during rendering.

`Graphics2D` provides several different ways to modify the transform in the `Graphics2D` context. The simplest is to call one of the `Graphics2D` transformation methods: `rotate`, `scale`, `shear`, or `translate`. You specify the characteristics of the transform that you want to be applied during rendering, and `Graphics2D` automatically makes the appropriate changes.

You can also explicitly *concatenate* an `AffineTransform` with the current `Graphics2D` transform. An `AffineTransform` performs a linear transformation such as translation, scaling, rotation, or shearing on a set of graphics primitives. When a transform is concatenated with an existing transform, the last transform specified is the *first* to be applied. To concatenate a transform with the current transform, you pass an `AffineTransform` to `Graphics2D.transform`.

`Graphics2D` also provides a version of `drawImage` that takes an `AffineTransform` as a parameter. This enables you to apply a transformation to an image object when it is drawn without permanently modifying the transformation pipeline. The image is drawn as if you had concatenated the transform with the current transform in the `Graphics2D` context.

Affine Transforms

The Java 2D API provides one transform class, `AffineTransform`. `AffineTransforms` are used to transform text, shapes, and images when they are rendered. You can also apply transforms to `Font` objects to create new font derivations, as discussed in "Creating Font Derivations" on page 65.

An affine transformation performs a linear transformation on a set of graphics primitives. It always transforms straight lines into straight lines and parallel lines into parallel lines; however, the distance between points and the angles between nonparallel lines might be altered.

Affine transformations are based on two-dimensional matrices of the following form:

$$\begin{bmatrix} a & c & t_x \\ b & d & t_y \end{bmatrix} \text{ where } x' = ax + cy + t_x \text{ and } y' = bx + dy + t_y$$

Transforms can be combined, effectively creating a series or *pipeline* of transformations that can be applied to an object. This combination is referred to as *concatenation*. When a transform is concatenated with an existing transform, such as with `AffineTransform.concatenate`, the last transform specified is the *first* to be applied. A transform can also be *pre-concatenated* with an existing transform. In this case, the last transform specified is the *last* to be applied.

Pre-concatenation is used to perform transformations relative to device space instead of user space. For example, you could use `AffineTransform.preConcatenate` to perform a translation relative to absolute pixel space.

2.2.6.1 Constructing an AffineTransform

`AffineTransform` provides a set of convenience methods for constructing `AffineTransform` objects:

- `getTranslateInstance`
- `getRotateInstance`
- `getScaleInstance`
- `getShearInstance`

To use these methods, you specify the characteristics of the transform you want to create and `AffineTransform` generates the appropriate transform matrix. You can also construct an `AffineTransform` by directly specifying the elements of the transformation matrix.

2.2.7 Composite Attributes

When two graphic objects overlap, it is necessary to determine what colors to render the overlapping pixels. For example, if a red rectangle and a blue rectangle overlap, the pixels that they share could be rendered red, blue, or some combination of the two. The color of the pixels in the overlapping area will determine which rectangle appears to be on top and how transparent it looks. The process of determining what color to render pixels shared by overlapping objects is called *compositing*.

Two interfaces form the basis of the Java 2D compositing model: `Composite` and `CompositeContext`.

To specify the compositing style that should be used, you add an `AlphaComposite` object to the `Graphics2D` context by calling `setComposite`. `AlphaComposite`, an implementation of the `Composite` interface, supports a number of different compositing styles. Instances of this class embody a compositing rule that describes how to blend a new color with an existing one.

One of the most commonly used compositing rules in the `AlphaComposite` class is `SRC_OVER`, which indicates that the new color (the source color) should be blended over the existing color (the destination color).

AlphaComposite Composition Rule	Description	Example
CLEAR	Clear	
DEST_IN	Destination In	
DEST_OUT	Destination Out	
DEST_OVER	Destination Over	
SRC	Source	
SRC_IN	Source In	
SRC_OUT	Source Out	
SRC_OVER	Source Over	

2.2.7.1 Managing Transparency

A color's *alpha* value is a measure of its transparency: it indicates, as a percentage, how much of a previously rendered color should show through when colors overlap. Opaque colors (`alpha=1.0`) don't allow any of the underlying color to show through, while transparent colors (`alpha=0.0`) let all of it show through.

When text and `Shapes` are rendered, the alpha value is derived from the `Paint` attribute in the `Graphics2D` context. When `Shapes` and text are antialiased, the alpha value from the `Paint` in the `Graphics2D` context is combined with pixel coverage information from the rasterized path. Images maintain their own alpha information—see "Transparency and Images" on page 26 for more information.

When you construct an `AlphaComposite` object, you can specify an additional alpha value. When you add this `AlphaComposite` object to the `Graphics2D` context, this extra alpha value increases the transparency of any graphic objects that are rendered—the alpha value of each graphic object is multiplied by the `AlphaComposite`'s alpha value.

2.2.7.2 Transparency and Images

Images can carry transparency information for each pixel in the image. This information, called an *alpha channel*, is used in conjunction with the `Composite` object in the `Graphics2D` context to blend the image with existing drawings.

For example, Figure 2-6 contains three images with different transparency information. In each case, the image is displayed over a blue rectangle. This example assumes that the `Graphics2D` context contains an `AlphaComposite` object that uses `SRC_OVER` as the compositing operation.

Figure 2-6 Transparency and Images

In the first image, all of the pixels are either fully opaque (the dog's body) or fully transparent (the background). This effect is often used on Web pages. In the second image, all of the pixels in the dog's body are rendered using a uniform, non-opaque alpha value, allowing the blue background to show through. In the third image, the pixels around the dogs face are fully opaque (alpha=1.0), but as the distance from its face increases, the alpha values for the pixels decrease.

2.3 Setting Up the Graphics2D Context

To configure the `Graphics2D` context for rendering, you use the `Graphics2D` set methods to specify attributes such as the `RenderingHints`, `Stroke`, `Paint`, clipping path, `Composite`, and `Transform`.

2.3.1 Setting Rendering Hints

A `RenderingHints` object encapsulates all of your preferences concerning how an object is rendered. To set the rendering hints in the `Graphics2D` context, you create a `RenderingHints` object and pass it into `Graphics2D.setRenderingHints`.

Setting a rendering hint does not guarantee that a particular rendering algorithm will be used: not all platforms support modification of the rendering mode.

In the following example, antialiasing is enabled and the rendering preference is set to quality:

```
qualityHints = new
               RenderingHints(RenderingHints.KEY_ANTIALIASING,
               RenderingHints.VALUE_ANTIALIAS_ON);
qualityHints.put(RenderingHints.KEY_RENDERING,
               RenderingHints.VALUE_RENDER_QUALITY);
g2.setRenderingHints(qualityHints);
```

2.3.2 Specifying Stroke Attributes

A `BasicStroke` defines the characteristics applied to a `Shape`'s outline, including its width and dashing pattern, how line segments are joined together, and the decoration (if any) applied to the end of a line. To set the stroke attributes in the

`Graphics2D` context, you create a `BasicStroke` object and pass it into `setStroke`.

2.3.2.1 Setting the Stroke Width

To set the stroke width, you create a `BasicStroke` object with the desired width and call `setStroke`.

In the following example, the stroke width is set to twelve points and the defaults are used for the join and endcap decorations:

```
wideStroke = new BasicStroke(12.0f);
g2.setStroke(wideStroke);
```

2.3.2.2 Specifying Join and Endcap Styles

To set the join and endcap styles, you create a `BasicStroke` object with the desired attributes.

In the following example, the stroke width is set to twelve points and the round join and endcap styles are used instead of the defaults:

```
roundStroke = new BasicStroke(4.0f, BasicStroke.CAP_ROUND,
              BasicStroke.JOIN_ROUND);
g2.setStroke(roundStroke);
```

2.3.2.3 Setting the Dashing Pattern

Complex dashing patterns can easily be defined with a `BasicStroke` object. When you create a `BasicStroke` object, you can specify two parameters that control the dashing pattern:

- `dash`—an array that represents the dashing pattern. Alternating elements in the array represent the dash size and the size of the space between dashes. Element 0 represents the first dash, element 1 represents the first space.
- `dash_phase`—an offset that defines where the dashing pattern starts.

In the following example, two different dashing patterns are applied to a line. In the first, the size of the dashes and the space between them is constant. The second dashing pattern is more complex, using a six-element array to define the dashing pattern.

```
float dash1[] = {10.0f};
BasicStroke bs = new BasicStroke(5.0f, BasicStroke.CAP_BUTT,
                 BasicStroke.JOIN_MITER, 10.0f, dash1, 0.0f);
g2.setStroke(bs);
Line2D line = new Line2D.Float(20.0f, 10.0f, 100.0f, 10.0f);
g2.draw(line);

float[] dash2 = {6.0f, 4.0f, 2.0f, 4.0f, 2.0f, 4.0f};
bs = new BasicStroke(5.0f, BasicStroke.CAP_BUTT,
     BasicStroke.JOIN_MITER, 10.0f, dash2, 0.0f);
g2.setStroke(bs);
g2.draw(line);
```

Both dashing patterns use a dash phase of zero, causing the dashes to be drawn starting at the beginning of the dashing pattern. The two dashing patterns are shown in Figure Figure 2-7.

Figure 2-7 Dashing Patterns

2.3.3 Specifying Fill Attributes

The `Paint` attribute in the `Graphics2D` context determines the fill color or pattern that is used when text and `Shapes` are rendered.

2.3.3.1 Filling a Shape with a Gradient

The `GradientPaint` class provides an easy way to fill a shape with a gradient of one color to another. When you create a `GradientPaint`, you specify a beginning position and color, and an ending position and color. The fill color changes proportionally from one color to the other along the line connecting the two positions, as shown in Figure 2-8.

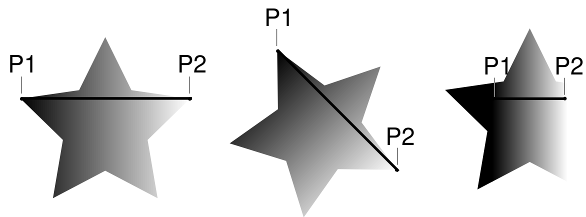


Figure 2-8 Creating Gradient Fills

In the third star in Figure 2-8, both points lie within the shape. All of the points along the gradient line extending beyond P1 take the beginning color, and the points along the gradient line extending beyond P2 take the ending color.

To fill a shape with a gradient of one color to another:

1. Create a `GradientPaint` object.
2. Call `Graphics2D.setPaint`.
3. Create the `Shape`.
4. Call `Graphics2D.fill(shape)`.

In the following example, a rectangle is filled with a blue-green gradient.

```
GradientPaint gp = new GradientPaint(50.0f, 50.0f, Color.blue
                     50.0f, 250.0f, Color.green);
g2.setPaint(gp);
g2.fillRect(50, 50, 200, 200);
```

2.3.3.2 Filling a Shape with a Texture

The `TexturePaint` class provides an easy way to fill a shape with a repeating pattern. When you create a `TexturePaint`, you specify a `BufferedImage` to use as the pattern. You also pass the constructor a rectangle to define the repetition frequency of the pattern, as shown in Figure 2-9.

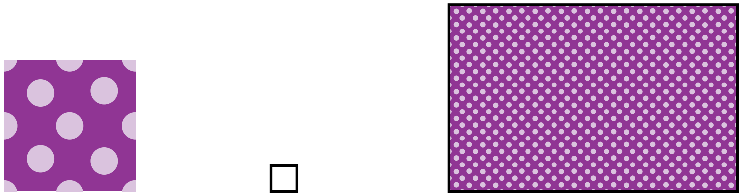


Figure 2-9 Creating Texture Paints

To fill a shape with a texture:

1. Create a `TexturePaint` object.
2. Call `Graphics2D.setPaint`.
3. Create the `Shape`.
4. Call `Graphics2D.fill(shape)`.

In the following example, a rectangle is filled with a simple texture created from a buffered image.

```
// Create a buffered image texture patch of size 5x5
BufferedImage bi = new BufferedImage(5, 5,
                       BufferedImage.TYPE_INT_RGB);
Graphics2D big = bi.createGraphics();
// Render into the BufferedImage graphics to create the texture
big.setColor(Color.green);
big.fillRect(0,0,5,5);
big.setColor(Color.lightGray);
big.fillOval(0,0,5,5);

// Create a texture paint from the buffered image
Rectangle r = new Rectangle(0,0,5,5);
TexturePaint tp = new
TexturePaint(bi,r,TexturePaint.NEAREST_NEIGHBOR);

// Add the texture paint to the graphics context.
g2.setPaint(tp);

// Create and render a rectangle filled with the texture.
```

```
g2.fillRect(0,0,200,200);
}
```

2.3.4 Setting the Clipping Path

To define a clipping path:

1. Create a `Shape` that represents the area you want to render.
2. Call `Graphics2D.setClip` to use the shape as the clipping path for the `Graphics2D` context.

To shrink the clipping path:

1. Create a `Shape` that intersects the current clipping path.
2. Call `clip` to change the clipping path to the intersection of the current clipping path and the new `Shape`.

In the following example, a clipping path is created from an ellipse and then modified by calling `clip`.

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

// The width and height of the canvas
  int w = getSize().width;
  int h = getSize().height;
  // Create an ellipse and use it as the clipping path
  Ellipse2D e = new Ellipse2D.Float(w/4.0f,h/4.0f,
                                    w/2.0f,h/2.0f);
  g2.setClip(e);

  // Fill the canvas. Only the area within the clip is rendered
  g2.setColor(Color.cyan);
  g2.fillRect(0,0,w,h);

 // Change the clipping path, setting it to the intersection of
  // the current clip and a new rectangle.
  Rectangle r = new Rectangle(w/4+10,h/4+10,w/2-20,h/2-20);
  g2.clip(r);
```

```
    // Fill the canvas. Only the area within the new clip
    // is rendered
    g2.setColor(Color.magenta);
    g2.fillRect(0,0,w,h);
  }
```

2.3.5 Setting the Graphics2D Transform

To transform a `Shape`, text string, or `Image` you add a new `AffineTransform` to the transformation pipeline in the `Graphics2D` context before rendering. The transformation is applied when the graphic object is rendered.

For example, to draw a rectangle that is rotated 45 degrees:

1. Get a rotation transform by calling `AffineTransform. getRotateInstance`.
2. Call `Graphics2D.setTransform` to add the new transform to the transformation pipeline.
3. Create a `Rectangle2D.Float` object.
4. Call `Graphics2D.draw` to render the rectangle.

In the following example, an instance of `AffineTransform` is used to rotate a rectangle 45 degrees when it is rendered.

```
Rectangle2D rect = new Rectangle2D.Float(1.0,1.0,2.0,3.0);
AffineTransform rotate45 =
  AffineTransform.getRotateInstance(Math.PI/4.0,0.0,0.0)
g2.setTransform(rotate45);
g2.draw(rect);
```

In this example, an `AffineTransform` is used to rotate a text string around a center point:

```
// Define the rendering transform
AffineTransform at = new AffineTransform();
// Apply a translation transform to make room for the
// rotated text.
at.setToTranslation(400.0, 400.0);
g2.transform(at);
```

```
// Create a rotation transform to rotate the text
at.setToRotation(Math.PI / 2.0);
// Render four copies of the string "Java" at 90 degree angles
for (int i = 0; i < 4; i++) {
    g2.drawString("Java", 0.0f, 0.0f);
    g2.transform(at);
}
```

You can transform an image in the same way—the transform in the `Graphics2D` context is applied during rendering regardless of the type of graphic object being rendered.

To apply a transform to an image *without* changing the transform in the `Graphics2D` context, you can pass an `AffineTransform` to `drawImage`:

```
AffineTransform rotate45 =
  AffineTransform.getRotateInstance(Math.PI/4.0,0.0,0.0)
g2.drawImage(myImage, rotate45);
```

Transforms can also be applied to a `Font` to create a modified version of the `Font`, for more information see "Creating Font Derivations" on page 65.

2.3.6 Specifying a Composition Style

An `AlphaComposite` encapsulates composition rules that determine how colors should be rendered when one object overlaps another. To specify the composition style for the `Graphics2D` context, you create an `AlphaComposite` and pass it into `setComposite`. The most commonly used is composition style is `SRC_OVER`.

2.3.6.1 Using the Source Over Compositing Rule

The `SRC_OVER` compositing rule composites the source pixel over the destination pixel such that the shared pixel takes the color of the source pixel. For example, if you render a blue rectangle and then render a red rectangle that partially overlaps it, the overlapping area will be red. In other words, the object that is rendered last will appear to be on top.

To use the `SRC_OVER` composition rule:

1. Create an `AlphaComposite` object by calling `getInstance` and specifying the

`SRC_OVER` rule.

```
AlphaComposite ac =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER);
```

2. Call `setComposite` to add the `AlphaComposite` object to the `Graphics2D` context.

```
g2.setComposite(ac);
```

Once the composite object is set, overlapping objects will be rendered using the specified composition rule.

2.3.6.2 Increasing the Transparency of Composited Objects

`AlphaComposite` allows you to specify an additional constant alpha value that is multiplied with the alpha of the source pixels to increase transparency.

For example, to create an `AlphaComposite` object that renders the source object 50% transparent, specify an alpha of .5:

```
AlphaComposite ac =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .5f);
```

In the following example, a source over alpha composite object is created with an alpha of .5 and added to the graphics context, causing subsequent shapes to be rendered 50% transparent.

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  g2.setColor(Color.red);
  g2.translate(100,50);
  // radians=degree * pie / 180
  g2.rotate((45*java.lang.Math.PI)/180);
  g2.fillRect(0,0,100,100);
  g2.setTransform(new AffineTransform());  // set to identity
  // Create a new alpha composite
  AlphaComposite ac =
     AlphaComposite.getInstance(AlphaComposite.SRC_OVER,0.5f);
  g2.setComposite(ac);
  g2.setColor(Color.green);
  g2.fillRect(50,0,100,100);
```

```
    g2.setColor(Color.blue);
    g2.fillRect(125,75,100,100);
    g2.setColor(Color.yellow);
    g2.fillRect(50,125,100,100);
    g2.setColor(Color.pink);
    g2.fillRect(-25,75,100,100);
}
```

2.4 Rendering Graphics Primitives

`Graphics2D` provides rendering methods for `Shapes`, `Text`, and `Images`:

- `draw`—strokes a `Shape`'s path using the `Stroke` and `Paint` objects in the `Graphics2D` context.
- `fill`—fills a `Shape` using the `Paint` in the `Graphics2D` context.
- `drawString`—renders the specified text string using the `Paint` in the `Graphics2D` context.
- `drawImage`—renders the specified image.

To stroke and fill a shape, you must call both the `draw` and `fill` methods.

`Graphics2D` also supports the draw and fill methods from previous versions of the JDK software, such as `drawOval` and `fillRect`.

2.4.1 Drawing a Shape

The outline of any `Shape` can be rendered with the `Graphics2D.draw` method. The draw methods from previous versions of the JDK software are also supported: `drawLine`, `drawRect`, `drawRoundRect`, `drawOval`, `drawArc`, `drawPolyline`, `drawPolygon`, `draw3DRect`.

When a `Shape` is drawn, its path is stroked with the `Stroke` object in the `Graphics2D` context. (See “Stroke Attributes” on page 19 for more information.) By setting an appropriate `BasicStroke` object in the `Graphics2D` context, you can draw lines of any width or pattern. The `BasicStroke` object also defines the line's endcap and join attributes.

To render shape's outline:

1. Create a `BasicStroke` object

2. Call `Graphics2D.setStroke`
3. Create the `Shape`.
4. Call `Graphics2D.draw(shape)`.

In the following example, a `GeneralPath` object is used to define a star and a `BasicStroke` object is added to the `Graphics2D` context to define the star's line with and join attributes.

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  // create and set the stroke
  g2.setStroke(new BasicStroke(4.0f));

  // Create a star using a general path object
  GeneralPath p = new GeneralPath(GeneralPath.NON_ZERO);
  p.moveTo(- 100.0f, - 25.0f);
  p.lineTo(+ 100.0f, - 25.0f);
  p.lineTo(- 50.0f, + 100.0f);
  p.lineTo(+ 0.0f, - 100.0f);
  p.lineTo(+ 50.0f, + 100.0f);
  p.closePath();

  // translate origin towards center of canvas
  g2.translate(100.0f, 100.0f);

  // render the star's path
  g2.draw(p);
}
```

2.4.2 Filling a Shape

The `Graphics2D.fill` method can be used to *fill* any `Shape`. When a `Shape` is filled, the area within its path is rendered with the `Graphics2D` context's current `Paint` attribute—a `Color`, `TexturePaint`, or `GradientPaint`.

The fill methods from previous versions of the JDK software are also supported: `fillRect`, `fill3DRect`, `fillRoundRect`, `fillOval`, `fillArc`, `fillPolygon`, `clearRect`.

To fill a `Shape`:

1. Set the fill color or pattern on the graphics context using `Graphics2D.setColor` or `Graphics2D.setPaint`.

1. Create the `Shape`.
2. Call `Graphics2D.fill` to render the `Shape`.

In the following example, `setColor` is called to define a green fill for a `Rectangle2D`.

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  g2.setPaint(Color.green);
  Rectangle2D r2 = new Rectangle2D.Float(25,25,150,150);

   g2.fill(r2);
}
```

2.4.3 Rendering Text

To render a text string, you call `Graphics2D.drawString`, passing in the string that you want to render. For more information about rendering text and selecting fonts, see "Fonts and Text Layout" on page 45.

2.4.4 Rendering Images

To render an `Image`, you create the `Image` and call `Graphics2D.drawImage`. For more information about processing and rendering images, see "Imaging" on page 67.

2.5 Defining Custom Composition Rules

You can create an entirely new type of compositing operation by implementing the `Composite` and `CompositeContext` interfaces. A `Composite` object provides a `CompositeContext` object that actually holds the state and performs the compositing work. Multiple `CompositeContext` objects can be created from one `Composite` object to maintain the separate states in a multithreaded environment.

3

Geometries

The Java 2D™ API provides several classes that define common geometric objects, such as points, lines, curves, and rectangles. These new geometry classes are part of the `java.awt.geom` package. For backward compatibility, the geometry classes that existed in previous versions of the JDK software, such as `Rectangle`, `Point`, and `Polygon`, remain in the `java.awt` package.

The Java 2D API geometries such as `GeneralPath`, `Arc2D`, and `Rectangle2D` implement the `Shape` interface defined in `java.awt`. `Shape` provides a common protocol for describing and inspecting geometric path objects. A new interface, `PathIterator`, defines methods for retrieving elements from a geometry.

Using the geometry classes, you can easily define and manipulate virtually any two-dimensional object.

3.1 Interfaces and Classes

The following tables list the key geometry interfaces and classes. Most of these interfaces and classes are part of the `java.awt.geom` package. Some, like `Shape`, are part of the `java.awt` package, primarily to maintain backward compatibility with earlier versions of the JDK software.

Interface	Description
`PathIterator`	Defines methods for retrieving elements from a path.
`Shape` (`java.awt`)	Provides a common set of methods for describing and inspecting geometric path objects. Implemented by `GeneralPath` and other geometry classes.

Class	Description

`Arc2D`
`Arc2D.Double`
`Arc2D.Float`
	Extends: `RectangularShape`
	Represents an arc defined by a bounding rectangle, start angle, angular extent, and a closure type. Implemented to specify arcs in float and double precision: `Arc2D.Float` and `Arc2D.Double`.

`Area`	Implements: `Shape, Cloneable` Represents an area geometry that supports boolean operations.

`CubicCurve2D`
`CubicCurve2D.Double`
`CubicCurve2D.Float`
	Implements: `Shape`
	Represents a cubic parametric curve segment in (w) coordinate space. Implemented to specify cubic curves in float and double precision: `CubicCurve2D.Float` and `CubicCurve2D.Double`.

`Dimension2D`	Encapsulates a width and height dimension. Abstract superclass for all objects that store a 2D dimension.

`Ellipse2D`
`Ellipse2D.Double`
`Ellipse2D.Float`
	Extends: `RectangularShape`
	Represents an ellipse defined by a bounding rectangle. Implemented to specify ellipses in float and double precision: `Ellipse2D.Float` and `Ellipse2D.Double`.

`FlatteningPathIterator`	Returns a flattened view of a `PathIterator` object. Can be used to provide flattening behavior for `Shapes` that don't perform the interpolation calculations themselves.
`GeneralPath`	Implements: `Shape` Represents a geometric path constructed from lines and quadratic and cubic curves.

`Line2D`
`Line2D.Double`
`Line2D.Float`
	Implements: `Shape`
	Represents a line segment in (x, y) coordinate space. Implemented to specify lines in float and double precision: `Line2D.Float` and `Line2D.Double`.

`Point2D`
`Point2D.Double`
`Point2D.Float`
	A point representing a location in (x,y) coordinate space. Implemented to specify points in float and double precision: `Point2D.Float` and `Point2D.Double`.

`QuadCurve2D`
`QuadCurve2D.Double`
`QuadCurve2D.Float`
	Implements: `Shape`
	Represents a quadratic parametric curve segment in (x, y) coordinate space. Implemented to specify quadratic curves in float and double precision: `QuadCurve2D.Float` and `QuadCurve2D.Double`.

`Rectangle2D`
`Rectangle2D.Double`
`Rectangle2D.Float`
	Extends: `RectangularShape`
	Represents a rectangle defined by a location (x, y) and dimension (w x h). Implemented to specify rectangles in float and double precision: `Rectangle2D.Float` and `Rectangle2D.Double`.

Class	Description
`RectangularShape`	Implements: `Shape` Provides common manipulation routines for operating on shapes that have rectangular bounds.

`RoundRectangle2D`
`RoundRectangle2D.Double`
`RoundRectangle2D.Float`
	Extends: `RectangularShape`
	Represents a rectangle with rounded corners defined by a location (x, y), a dimension (w x h), and the width and height of the corner arc. Implemented to specify round rectangles in float and double precision: `RoundRectangle2D.Float` and `RoundRectangle2D.Double`.

3.2 Geometry Concepts

A `Shape` is an instance of any class that implements the `Shape` interface, such as `GeneralPath` or `Rectangle2D.Float`. A `Shape`'s contour (outline) is referred to as its *path*.

When a `Shape` is drawn, the pen style defined by the `Stroke` object in the `Graphics2D` context is applied to the `Shape`'s path. When a `Shape` is filled, the `Paint` in the `Graphics2D` context is applied to the area within its path. For more information, see "Rendering with Graphics2D" on page 15.

A `Shape`'s path can be also used to define a *clipping path*. A clipping path determines what pixels are rendered—only those pixels that lie within the area defined by the clipping path are rendered. The clipping path is part of the `Graphics2D` context. For more information, see "Setting the Clipping Path" on page 32.

A `GeneralPath` is a shape that can be used to represent any two-dimensional object that can be constructed from lines and quadratic or cubic curves. For convenience, `java.awt.geom` provides additional implementations of the `Shape` interface that represent common geometric objects such as rectangles, ellipses, arcs, and curves. The Java2D™ API also provides a special type of shape that supports constructive area geometry.

3.2.1 Constructive Area Geometry

Constructive Area Geometry (CAG) is the process of creating new geometric objects by performing boolean operations on existing objects. In the Java 2D API, a special type of `Shape` called an `Area` supports boolean operations. You can construct an `Area` from any `Shape`.

`Areas` support the following Boolean operations:

- Union

- Intersection
- Subtraction
- Exclusive OR (XOR)

These operations are illustrated in Figure 3-1.

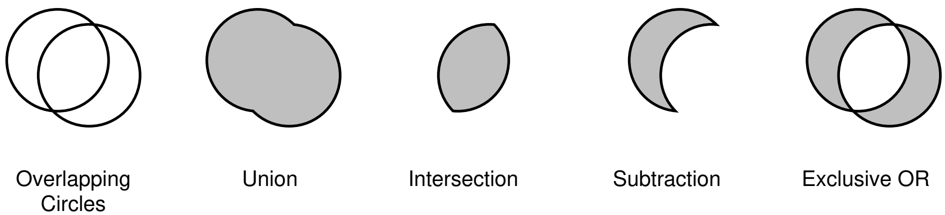


Figure 3-1 Boolean Operations

3.2.2 Bounds and Hit Testing

A *bounding box* is a rectangle that fully encloses a shape's geometry. Bounding boxes are used to determine whether or not an object has been selected or "hit" by the user.

The `Shape` interface defines two methods for retrieving a shape's bounding box, `getBounds` and `getBounds2D`. The `getBounds2D` method returns a `Rectangle2D` instead of a `Rectangle`, providing a higher-precision description of the shape's bounding box.

`Shape` also provides methods for determining whether or not:

- A specified point lies within the bounds of the shape (`contains`)
- A specified rectangle lies totally within the bounds of the shape (`contains`)
- A specified rectangle intersects the shape (`intersects`)

3.3 Combining Areas to Create New Shapes

`Areas` can be used to quickly construct complex `Shapes` from simple shapes such as circles and squares. To create a new complex `Shape` by combining `Areas`:

1. Using `Shapes`, construct the `Areas` to be combined.

2. Call the appropriate Boolean operators: `add`, `subtract`, `intersect`, `exclusiveOr`.

For example, CAG could be used to create a pear like that shown in `Figure 3-2`.

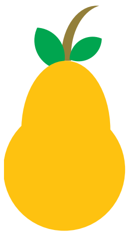


Figure 3-2 Pear constructed from circles

The body of the pear is constructed by performing a union operation on two overlapping `Areas`: a circle and an oval. The leaves are each created by performing an intersection on two overlapping circles and then joined into a single `Shape` through a union operation. Overlapping circles are also used to construct the stem through two subtraction operations.

3.4 Creating a Custom Shape

You can implement the `Shape` interface to create a class that defines a new type of shape. It doesn't matter how you represent the shape internally, as long as you can implement the `Shape` interface methods. The `Shape` must be able to generate a path that specifies its contour.

For example, you could create a simple implementation of `Shape` that represents polygons as arrays of points. Once the polygon is built, it could be passed to `draw`, `setClip`, or any other method that expects a `Shape` object as an argument.

The `PolygonPath` class must implement the `Shape` interface methods:

- `contains`
- `getBounds`
- `getBounds2D`
- `getPathIterator`
- `intersects`

4

Fonts and Text Layout

You can use the Java 2D™ API transformation and drawing mechanisms with text strings. In addition, the Java 2D API provides text-related classes that support fine-grain font control and sophisticated text layout. These include an enhanced `Font` class and the new `TextLayout` class.

This chapter focuses on the new font and text layout capabilities supported through interfaces and classes in `java.awt`, and `java.awt.font`. **For more information about using these features, see the 2D Text Tutorial that's available through the Java Developer Connection at http://developer.java.sun.com/developer/onlineTraining/Graphics/2DText/.**

For information about text analysis and internationalization, refer to the `java.text` documentation and the "Writing Global Programs" track in the Java Tutorial. For information about using the text layout mechanisms implemented in Swing, see the `java.awt.swing.text` documentation and "Using the JFC/Swing Packages" in the Java Tutorial.

Note: The information on international text layout contained in this chapter is based on the paper *International Text in JDK 1.2* by Mark Davis, Doug Felt, and John Raley, copyright 1997, Taligent, Inc.

4.1 Interfaces and Classes

The following tables list the key font and text layout interfaces and classes. Most of these interfaces and classes are part of the `java.awt.font` package. Some, like `Font`, are part of the `java.awt` package to maintain backward compatibility with earlier versions of the JDK.

Interface	Description
`MultipleMaster`	Represents Type 1 Multiple Master fonts. Implemented by `Font` objects that are multiple master fonts to enable access to multiple master design controls.
`OpenType`	Represents Open Type and True Type fonts. Implemented by `Font` objects that are Open Type or True Type fonts to enable access to the font's `sfnt` tables.

Class	Description
`Font` (`java.awt`)	Represents an instance of a font face from the collection of font faces available on the host system. Supports the specification of detailed font information and provides access to information about the font and its glyphs.
`FontRenderContext`	Encapsulates the information necessary to correctly measure text.
`GlyphJustificationInfo`	Represents information about the justification properties of a glyph, such as weight, priority, absorb, and limit.
`GlyphMetrics`	Provides metrics for a single glyph.
`GlyphVector`	A collection of glyphs and their positions.
`GraphicAttribute`	Base class for a `TextLayout` attribute that specifies a graphic to be embedded within text. Implemented by `ShapeGraphicAttribute` and `ImageGraphicAttribute`, which enable `Shapes` and `Images` to be embedded in a `TextLayout`. Can be subclassed to implement custom character replacement graphics.
`ImageGraphicAttribute`	Extends: `GraphicAttribute` A `GraphicsAttribute` used to draw `Images` within a `TextLayout`.
`LineBreakMeasurer`	Breaks a block of text that spans multiple lines into `TextLayout` objects that fit within a specified line length.
`LineMetrics`	Provides access to the font metrics needed to lay out characters along a line and to lay out a set of lines. These metrics include ascent, descent, leading, height, and baseline information.
`ShapeGraphicAttribute`	Extends: `GraphicAttribute` A `GraphicsAttribute` used to draw `Shapes` within a `TextLayout`.
`TextAttribute`	Defines attribute keys and values used for text rendering.
`TextHitInfo`	Represents hit test information for characters in a `TextLayout`.

Class	Description
`TextLayout`	Implements: `Cloneable` Provides an immutable graphical representation of styled character data, including bidirectional text.

4.2 Font Concepts

The `Font` class has been enhanced to support the specification of detailed font information and enable the use of sophisticated typographic features.

A `Font` object represents an instance of a font face from the collection of font faces available on the system. Examples of common font faces include `Helvetica Bold` and `Courier Bold Italic`.

Three names are associated with a `Font`—its logical name, family name, and font face name:

- A `Font` object's *logical name* is a name mapped onto one of the specific fonts available on the platform. The logical font name is the name used to specify a `Font` in JDK 1.1 and earlier releases. When specifying a `Font` in Java™ 2 SDK, you should use the *font face name* instead of the logical name.You can get the logical name from the `Font` by calling `getName`. To get a list of the logical names that are mapped onto the specific fonts available on a platform, call `java.awt.Toolkit.getFontList`.
- A `Font` object's *family name* is the name of the font family that determines the typographic design across several faces, such as Helvetica. You retrieve the family name through the `getFamily` method.
- A `Font` objects' *font face name* refers to an actual font installed on the system. This is the name you should use when specifying a font in Java 2 SDK. It's often referred to as just the *font name*. You can retrieve the font name by calling `getFontName`. To determine which font faces are available on the system, you can call `GraphicsEnvironment.getAllFonts`.

You can access information about a `Font` through the `getAttributes` method. A `Font`'s attributes include its name, size, transform, and font features such as weight and posture.

A `LineMetrics` object encapsulates the measurement information associated with a `Font`, such as its ascent, descent, and leading:

- A*scent* is the distance from the baseline to the ascender line. This distance represents the typical height of capital letters, but some characters might extend above the ascender line.
- *Descent* is the distance from the baseline to the descender line. The lowest point of most characters will fall within the descent, but some characters might extend below the descender line.
- *Leading* is the recommended distance from the bottom of the descender line to the top of the next line.

Figure 4-1 Line Metrics

This information is used to properly position characters along a line, and to position lines relative to one another. You can access these line metrics through the `getAscent`, `getDescent`, and `getLeading` methods. You can also access information about a `Font`'s height, baseline, and underline and strikethrough characteristics through `LineMetrics`.

4.3 Text Layout Concepts

Before a piece of text can be displayed, it must be properly shaped and positioned using the appropriate glyphs and ligatures. This process is referred to as *text layout*. The text layout process involves:

- Shaping text using the appropriate glyphs and ligatures.
- Properly ordering the text.
- Measuring and positioning the text.

The information used to lay out text is also necessary for performing text operations such as caret positioning, hit detection, and highlighting.

To develop software that can be deployed in international markets, text must be laid out in different languages in a way that conforms to the rules of the appropriate writing system.

4.3.1 Shaping Text

A *glyph* is the visual representation of one or more characters. The shape, size, and position of a glyph is dependent on its context. Many different glyphs can be used to represent a single character or combination of characters, depending on the font and style.

For example, in handwritten cursive text, a particular character can take on different shapes depending on how it is connected to adjacent characters.

In some writing systems, particularly Arabic, the context of a glyph must always be taken into account. Unlike in English, cursive forms are mandatory in Arabic; it is unacceptable to present text without using cursive forms.

Depending on the context, these cursive forms can differ radically in shape. For example, the Arabic letter *heh* has the four cursive forms shown in Figure 4-2.

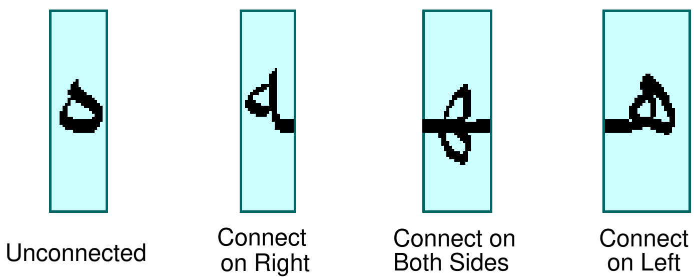


Figure 4-2 Cursive Forms in Arabic

Although these four forms are quite different from one another, such cursive shape-changing is not fundamentally different from cursive writing in English.

In some contexts, two glyphs can change shape even more radically and merge to form a single glyph. This type of merged glyph is called a *ligature*. For example, most English fonts contain the ligature *fi* shown in Figure 4-3. The merged glyph takes into account the overhang on the letter *f* and combines the characters in a natural-looking way, instead of simply letting the letters collide.

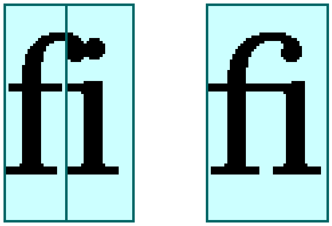


Figure 4-3 English Ligatures

Ligatures are also used in Arabic and the use of some ligatures is mandatory—it is unacceptable to present certain character combinations without using the appropriate ligature. When ligatures are formed from Arabic characters, the shapes change even more radically than they do in English. For example, Figure 4-4 illustrates how two Arabic characters are combined into a single ligature when they appear together.

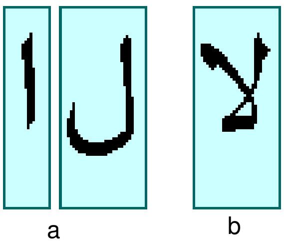


Figure 4-4 Arabic Ligatures

4.3.2 Ordering Text

In the Java™ programming language, text is encoded using Unicode character encoding. Text that uses Unicode character encoding is stored in memory in *logical order*. Logical order is the order in which characters and words are read and written. The logical order is not necessarily the same as the *visual order*, the order in which the corresponding glyphs are displayed.

The visual order for glyphs in a particular writing system (script) is called the *script order*. For example, the script order for Roman text is left-to-right and the script order for Arabic and Hebrew is right-to-left.

Some writing systems have rules in addition to script order for arranging glyphs and words on lines of text. For example, Arabic and Hebrew numbers run left to right, even though the letters run right to left. (This means that Arabic and Hebrew, even with no embedded English text, are truly bidirectional.)

A writing system's visual order must be maintained even when languages are mixed together. This is illustrated in Figure 4-5, which displays an Arabic phrase embedded in an English sentence.

Note: In this and subsequent examples, Arabic and Hebrew text is represented by uppercase letters and spaces are represented by underscores. Each illustration contains two parts: a representation of the characters stored in memory (the characters in logical order) followed by a representation of how those characters are displayed (the characters in visual order). The numbers below the character boxes indicate the insertion offsets.

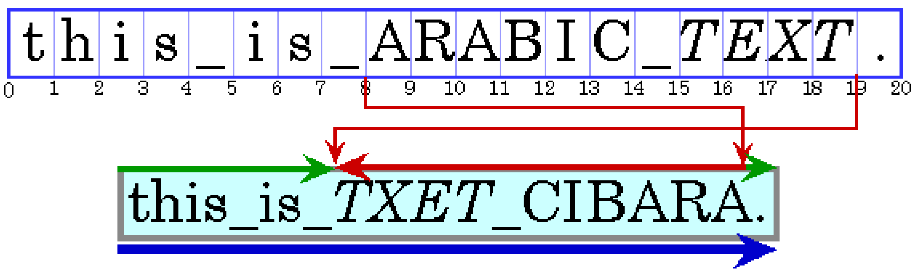


Figure 4-5 Bidirectional Text

Even though they are part of an English sentence, the Arabic words are displayed in the Arabic script order, right-to-left. Because the italicized Arabic word is logically after the Arabic in plain text, it is visually to the left of the plain text.

When a line with a mixture of left-to-right and right-to-left text is displayed, the *base direction* is significant. The base direction is the script order of the predominant writing system. For example, if the text is primarily English with some embedded Arabic, then the base direction is left-to-right. If the text is primarily Arabic with some embedded English or numbers, then the base direction is right-to-left.

The base direction determines the order in which segments of text with a common direction are displayed. In the example shown in Figure 4-5, the base direction is left-to-right. There are three directional runs in this example: the English text at the beginning of the sentence runs left to right, the Arabic text runs right to left, and the period runs left to right.

Graphics are often embedded in the flow of text. These inline graphics behave like glyphs in terms of how they affect the text flow and line wrapping. Such inline graphics need to be positioned using the same bidirectional layout algorithm so that they appear in the proper location in the flow of characters.

For more information about the precise algorithm used to order glyphs within a line, see the description of the Bidirectional Algorithm in The Unicode Standard, Version 2.0, Section 3.11.

4.3.3 Measuring and Positioning Text

Unless you are working with a monospace font, different characters in a font have different widths. This means that all positioning and measuring of text has to take into account exactly which characters are used, not just how many. For example, to right-align a column of numbers displayed in a proportional font, you can't simply use extra spaces to position the text. To properly align the column, you need to know the exact width of each number so that you can adjust accordingly.

Text is often displayed using multiple fonts and styles, such as bold or italic. In this case, even the same character can have different shapes and widths, depending on how it is styled. To properly position, measure, and render text, you need to keep track of each individual character *and* the style applied to that character. Fortunately, `TextLayout` does this for you.

To properly display text in languages such as Hebrew and Arabic, each individual character needs to be measured and positioned within the context of neighboring characters. Because the shapes and positions of the characters can change depending on the context, measuring and positioning such text without taking the context into account produces unacceptable results.

4.3.4 Supporting Text Manipulation

To allow the user to edit the text that is displayed, you must be able to:

- Display a caret that indicates where new characters will be inserted when the user enters text.
- Move the caret and insertion point in response to user input.
- Detect user selections (hit detection).
- Highlight selected text.

4.3.4.1 Displaying Carets

In editable text, a *caret* is used to graphically represent the current insertion point, the position in the text where new characters will be inserted. Typically, a caret is shown as a blinking vertical bar between two glyphs. New characters are inserted and displayed at the caret's location.

Calculating the caret position can be complicated, particularly for bidirectional text. Insertion offsets on directional boundaries have two possible caret positions because the two glyphs that correspond to the character offset are not displayed adjacent to one another. This is illustrated in Figure 4-6. In this figure, the carets are shown as square brackets to indicate the glyph to which the caret corresponds.

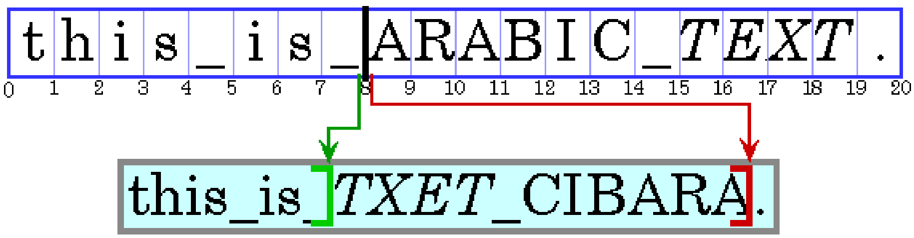


Figure 4-6 Dual Carets

Character offset 8 corresponds to the location after the _ and before the *A*. If the user enters an Arabic character, its glyph is displayed to the right of (before) the *A*; if the user enters an English character, its glyph is displayed to the right of (after) the _.

To handle this situation, some systems display dual carets, a strong (primary) caret and a weak (secondary) caret. The strong caret indicates where an inserted character will be displayed when that character's direction is the same as the base direction of the text. The weak caret shows where an inserted character will be displayed when the character's direction is the opposite of the base direction. `TextLayout` automatically supports dual carets; `JTextComponent` does not.

When you're working with bidirectional text, you can't simply add the widths of the glyphs before a character offset to calculate the caret position. If you did, the caret would be drawn in the wrong place, as shown in Figure 4-7.

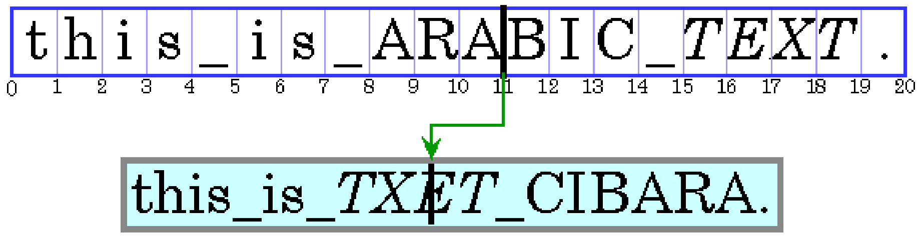


Figure 4-7 Caret Drawn Incorrectly

For the caret to be properly positioned, the widths of the glyphs to the left of the offset need to be added and the current context taken into account. Unless the context is taken into account, the glyph metrics won't necessarily match the display. (The context can affect which glyphs are used.)

4.3.4.2 Moving Carets

All text editors allow the user to move the caret with the arrow keys. Users expect the caret to move in the direction of the pressed arrow key. In left-to-right text, moving the insertion offset is simple: the right arrow key increases the insertion offset by one and the left arrow key decreases it by one. In bidirectional text or in text with ligatures, this behavior would cause the caret to jump across glyphs at direction boundaries and move in the reverse direction within different directional runs.

To move the caret smoothly through bidirectional text, you need to take into account the direction of the text runs. You can't simply increment the insertion offset when the right arrow key is pressed and decrement it when the left arrow key is pressed. If the current insertion offset is within a run of right-to-left characters, the right arrow key should decrease the insertion offset, and the left arrow key should increase it.

Moving the caret across a directional boundary is even more complicated. Figure 4-8 illustrates what happens when a directional boundary is crossed when the user is navigating with the arrow key. Stepping three positions to the right in the displayed text corresponds to moving to the character offsets 7, 19, then 18.

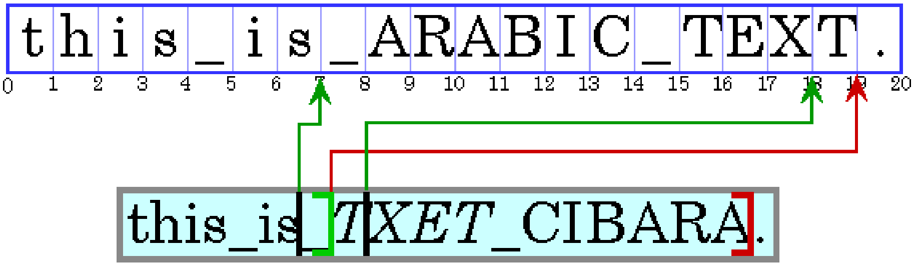


Figure 4-8 Caret Movement

Certain glyphs should never have a caret between them; instead, the caret should move as though the glyphs represented a single character. For example, there should never be a caret between an *o* and an umlaut if they are represented by two separate characters. (See The Unicode Standard, Version 2.0, Chapter 5, for more information.)

`TextLayout` provides methods (`getNextRightHit` and `getNextLeftHit`) that enable you to easily move the caret smoothly through bidirectional text.

4.3.4.3 Hit Testing

Often, a location in device space must be converted to a text offset. For example, when a user clicks the mouse on selectable text, the location of the mouse is converted to a text offset and used as one end of the selection range. Logically, this is the inverse of positioning a caret.

When you're working with bidirectional text, a single visual location in the display can correspond to two different offsets in the source text, as shown in Figure 4-9.

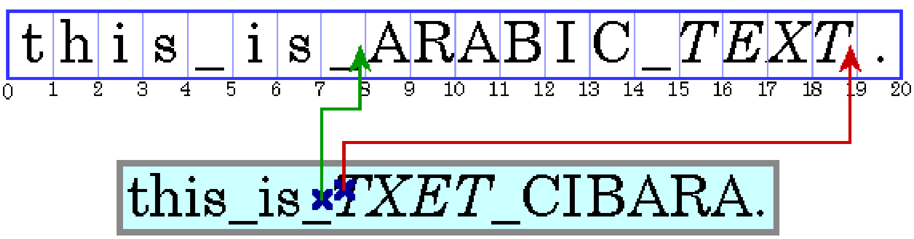


Figure 4-9 Hit Testing Bidirectional Text

Because a single visual location can correspond to two different offsets, hit testing bidirectional text isn't just a matter of measuring glyph widths until the glyph at the correct location is found and then mapping that position back to a character offset. Detecting the side that the hit was on helps distinguish between the two alternatives.

You can perform hit testing using `TextLayout.hitTestChar`. Hit information is encapsulated in a `TextHitInfo` object and includes information about the side that the hit was on.

4.3.4.4 Highlighting Selections

A selected range of characters is represented graphically by a highlight region, an area in which glyphs are displayed with inverse video or against a different background color.

Highlight regions, like carets, are more complicated for bidirectional text than for monodirectional text. In bidirectional text, a contiguous range of characters might not have a contiguous highlight region when displayed. Conversely, a highlight region showing a visually contiguous range of glyphs might not correspond to a single, contiguous range of characters.

This results in two strategies for highlighting selections in bidirectional text:

- *Logical highlighting* —with logical highlighting, the selected characters are always contiguous in the text model, and the highlight region is allowed to be discontiguous. For an example of logical highlighting, see Figure 4-10.
- *Visual highlighting*—with visual highlighting, there might be more than one range of selected characters, but the highlight region is always contiguous. For an example of visual highlighting, see Figure 4-11.

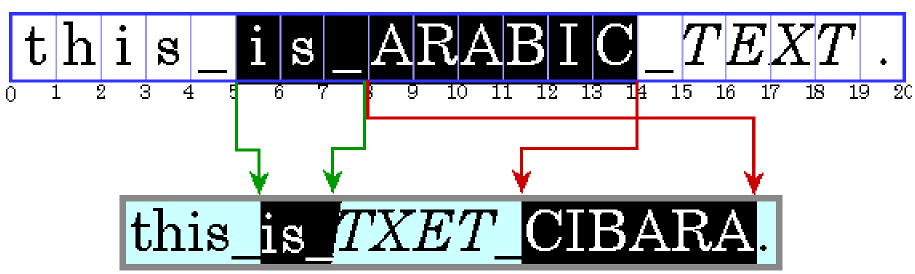


Figure 4-10 Logical Highlighting (contiguous characters)

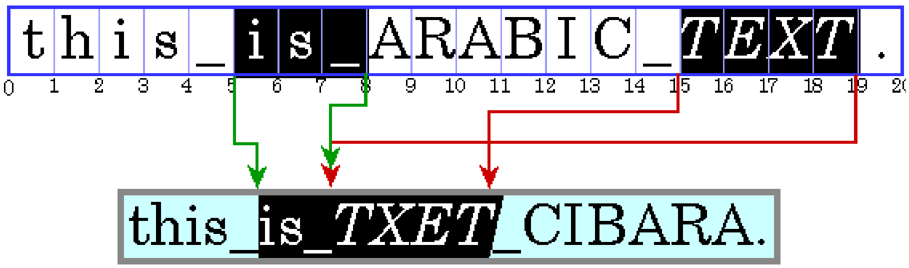


Figure 4-11 Visual Highlighting (contiguous highlight region)

Logical highlighting is simpler to implement, since the selected characters are always contiguous in the text.

4.3.5 Performing Text Layout in a Java™ Application

Depending on which Java™ APIs you use, you can have as little or as much control over text layout as you need:

- If you just want to display a block of text or need an editable text control, you can use `JTextComponent`, which will perform the text layout for you. `JTextComponent` is designed to handle the needs of most international applications and supports bidirectional text For more information about `JTextComponent`, see "Using the JFC/Swing Packages" in the Java Tutorial.
- If you want to display a simple text string, you can call `Graphics2D.drawString` and let Java 2D™ lay out the string for you. You can also use `drawString` to render styled strings and strings that contain bidirectional text. For more information about rendering text through `Graphics2D`, see "Rendering Graphics Primitives" on page 36.
- If you want to implement your own text editing routines, you can use `TextLayout` to manage text layout, highlighting, and hit detection. The facilities provided by `TextLayout` handle most common cases, including text strings with mixed fonts, mixed languages, and bidirectional text. For more information about using TextLayout, see "Managing Text Layout" on page 58.
- If you want total control over how text is shaped and positioned, you can construct your own `GlyphVectors` using `Font` and then render them through `Graphics2D`. For more information about implementing your own text layout mechanism, see "Implementing a Custom Text Layout Mechanism" on page 63.

Generally, you do not need to perform text layout operations yourself. For most applications, `JTextComponent` is the best solution for displaying static and editable text. However, `JTextComponent` does not support the display of dual carets or discontiguous selections in bidirectional text. If your application requires these features, or you prefer to implement your own text editing routines, you can use the Java 2D text layout APIs.

4.4 Managing Text Layout

The `TextLayout` class supports text that contains multiple styles and characters from different writing systems, including Arabic and Hebrew. (Arabic and Hebrew are particularly difficult to display because you must reshape and reorder the text to achieve an acceptable representation.)

`TextLayout` simplifies the process of displaying and measuring text even if you are working with English-only text. By using `TextLayout`, you can achieve high-quality typography with no extra effort.

Text Layout Performance

`TextLayout` is designed so that there is no significant performance impact when it's used to display simple, monodirectional text. There is some additional processing overhead when `TextLayout` is used to display Arabic or Hebrew text. However, it's typically on the order of microseconds per character and is dominated by the execution of normal drawing code.

The `TextLayout` class manages the positioning and ordering of glyphs for you. You can use `TextLayout` to:

- Lay out monodirectional and bidirectional text
- Display and move carets
- Perform hit testing on text
- Highlight text selections

In some situations, you might want to compute the text layout yourself, so that you can control exactly which glyphs are used and where they are placed. Using information such as glyph sizes, kerning tables, and ligature information, you can construct your own algorithms for computing the text layout, bypassing the system's layout mechanism. For more information, see "Implementing a Custom Text Layout Mechanism" on page 63.

4.4.1 Laying Out Text

`TextLayout` automatically lays out text, including bidirectional (BIDI) text, with the correct shaping and ordering. To correctly shape and order the glyphs representing a line of text, `TextLayout` must know the full context of the text:

- If the text fits on a single line, such as a single-word label for a button or a line in a dialog box, you can construct a `TextLayout` directly from the text.
- If you have more text than can fit on a single line or want to break text on a single line into tabbed segments, you cannot construct a `TextLayout` directly. You must use a `LineBreakMeasurer` to provide sufficient context.

The base direction of the text is normally set by an attribute (style) on the text. If that attribute is missing, `TextLayout` follows the Unicode bidirectional algorithm and derives the base direction from the initial characters in the paragraph.

4.4.2 Displaying Dual Carets

`TextLayout` maintains caret information such as the caret `Shape`, position, and angle. You can use this information to easily display carets in both monodirectional and bidirectional text. When you're drawing carets for bidirectional text, using `TextLayout` ensures that the carets will be positioned correctly.

`TextLayout` provides default caret `Shapes` and automatically supports dual carets. For italic and oblique glyphs, `TextLayout` produces angled carets, as shown in Figure 4-12. These caret positions are also used as the boundaries between glyphs for highlighting and hit testing, which helps produce a consistent user experience.

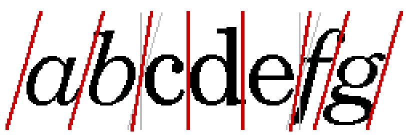


Figure 4-12 Angled Carets

Given an insertion offset, the `getCaretShapes` method returns a two-element array of `Shapes`: element 0 contains the strong caret and element 1 contains the weak caret, if one exists. To display dual carets, you simply draw both caret `Shapes`; the carets will be automatically be rendered in the correct positions.

If you want to use custom caret `Shapes`, you can retrieve the position and angle of the carets from the `TextLayout` and draw them yourself.

In the following example, the default strong and weak caret `Shapes` are drawn in different colors. This is a common way to differentiate dual carets.

```
Shape[] caretShapes = layout.getCaretShapes(hit);
g2.setColor(PRIMARY_CARET_COLOR);
g2.draw(caretShapes[0]);
if (caretShapes[1] != null){
  g2.setColor(SECONDARY_CARET_COLOR);
  g2.draw(caretShapes[1]);
}
```

4.4.3 Moving the Caret

You can also use `TextLayout` to determine the resulting insertion offset when a user presses the left or right arrow key. Given a `TextHitInfo` object that represents the current insertion offset, the `getNextRightHit` method returns a `TextHitInfo` object that represents the correct insertion offset if the right arrow key is pressed. The `getNextLeftHit` method provides the same information for the left arrow key.

In the following example, the current insertion offset is moved in response to a right arrow key.

```
TextHitInfo newInsertionOffset =
            layout.getNextRightHit(insertionOffset);
if (newInsertionOffset != null) {
  Shape[] caretShapes =
          layout.getCaretShapes(newInsertionOffset);
  // draw carets
  ...
  insertionOffset = newInsertionOffset;
}
```

4.4.4 Hit Testing

`TextLayout` provides a simple mechanism for hit testing text. The `hitTestChar` method takes *x* and *y* coordinates from the mouse as arguments and returns a `TextHitInfo` object. The `TextHitInfo` contains the insertion offset for the specified position and the side that the hit was on. The insertion offset is the offset closest to the hit: if the hit is past the end of the line, the offset at the end of the line is returned.

In the following example, `hitTestChar` is called on a `TextLayout` and then `getInsertIndex` is used to retrieve the offset.

```
TextHitInfo hit = layout.hitTestChar(x, y);
int insertIndex = hit.getInsertIndex();
```

4.4.5 Highlighting Selections

You can get a `Shape` that represents the highlight region from the `TextLayout`. `TextLayout` automatically takes the context into account when calculating the dimensions of the highlight region. `TextLayout` supports both logical and visual highlighting.

In the following example, the highlight region is filled with the highlight color and then the `TextLayout` is drawn over the filled region. This is one simple way to display highlighted text.

```
Shape highlightRegion = layout.getLogicalHighlightShape(hit1,
      hit2);
graphics.setColor(HIGHLIGHT_COLOR);
graphics.fill(highlightRegion);
graphics.drawString(layout, 0, 0);
```

4.4.6 Querying Layout Metrics

`TextLayout` provides access to graphical metrics for the entire range of text it represents. Metrics available from `TextLayout` include the ascent, descent, leading, advance, visible advance, and the bounding rectangle.

More than one `Font` can be associated with a `TextLayout`: different style runs can use different fonts. The ascent and descent values for a `TextLayout` are the maximum values of all of the fonts used in the `TextLayout`. The computation of the `TextLayout`'s leading is more complicated; it's not just the maximum leading value.

The advance of a `TextLayout` is its length: the distance from the left edge of the leftmost glyph to the right edge of the rightmost glyph. The advance is sometimes referred to as the *total advance*. The *visible advance* is the length of the `TextLayout` without its trailing whitespace.

The bounding box of a `TextLayout` encloses all of the text in the layout. It includes all the visible glyphs and the caret boundaries. (Some of these might hang over the origin or origin + advance). The bounding box is relative to the origin of the `TextLayout`, not to any particular screen position.

In the following example, the text in a `TextLayout` is drawn within the layout's bounding box.

```
graphics.drawString(layout, 0, 0);
Rectangle2D bounds = layout.getBounds();
graphics.drawRect(bounds.getX()-1, bounds.getY()-1,
         bounds.getWidth()+2, bounds.getHeight()+2);
```

4.4.7 Drawing Text Across Multiple Lines

`TextLayout` can also be used to display a piece of text that spans multiple lines. For example, you might take a paragraph of text, line-wrap the text to a certain width, and display the paragraph as multiple lines of text.

To do this, you do not directly create the `TextLayouts` that represent each line of text—`LineBreakMeasurer` generates them for you. Bidirectional ordering cannot always be performed correctly unless all of the text in a paragraph is available. `LineBreakMeasurer` encapsulates enough information about the context to produce correct `TextLayouts`.

When text is displayed across multiple lines, the length of the lines is usually determined by the width of the display area. Line breaking (line wrapping) is the process of determining where lines begin and end, given a graphical width in which the lines must fit.

The most common strategy is to place as many words on each line as will fit. This strategy is implemented in `LineBreakMeasurer`. Other more complex line break strategies use hyphenation, or attempt to minimize the differences in line length within paragraphs. The Java 2D™ API does not provide implementations of these strategies.

To break a paragraph of text into lines, you construct a `LineBreakMeasurer` with the entire paragraph and then call `nextLayout` to step through the text and generate `TextLayouts` for each line.

To do this, `LineBreakMeasurer` maintains an offset within the text. Initially, the offset is at the beginning of the text. Each call to `nextLayout` moves the offset by the character count of the `TextLayout` that was created. When this offset reaches the end of the text, `nextLayout` returns `null`.

The visible advance of each `TextLayout` that the `LineBreakMeasurer` creates doesn't exceed the specified line width. By varying the width you specify when you call `nextLayout`, you can break text to fit complicated areas, such as an HTML page with images in fixed positions or tab-stop fields. You can also pass in a `BreakIterator` to tell `LineBreakMeasurer` where valid breakpoints are; if you don't supply one the `BreakIterator` for the default locale is used.

In the following example, a bilingual text segment is drawn line by line. The lines are aligned to either to the left margin or right margin, depending on whether the base text direction is left-to-right or right-to-left.

```
Point2D pen = initialPosition;
LineBreakMeasurer measurer = new LineBreakMeasurer(styledText,
myBreakIterator);
while (true) {
  TextLayout layout = measurer.nextLayout(wrappingWidth);
  if (layout == null) break;
    pen.y += layout.getAscent();
    float dx = 0;
    if (layout.isLeftToRight())
      dx = wrappingWidth - layout.getAdvance();
    layout.draw(graphics, pen.x + dx, pen.y);
    pen.y += layout.getDescent() + layout.getLeading();
}
```

4.5 Implementing a Custom Text Layout Mechanism

The `GlyphVector` class provides a way to display the results of custom layout mechanisms. A `GlyphVector` object can be thought of as the output of an algorithm that takes a string and computes exactly how the string should be displayed. The system has a built-in algorithm and the Java 2D™ API lets advanced clients define their own algorithms.

A `GlyphVector` object is basically an array of glyphs and glyph locations. Glyphs are used instead of characters to provide total control over layout characteristics such as kerning and ligatures. For example, when displaying the string "`final`", you might want to replace the leading `fi` substring with the ligature *ﬁ*. In this case, the `GlyphVector` object will have fewer glyphs than the number of characters in the original string.

Figure 4-13 and Figure 4-14 illustrate how `GlyphVector` objects are used by layout mechanisms. Figure 4-13 shows the default layout mechanism. When `drawString` is called on a `String`, the built-in layout algorithm:

- Uses the current `Font` in the `Graphics2D` context to determine which glyphs to use.
- Calculates where each glyph should be placed.
- Stores the resulting glyph and position information in a `GlyphVector`.
- Passes the `GlyphVector` to a glyph rendering routine that does the actual drawing.

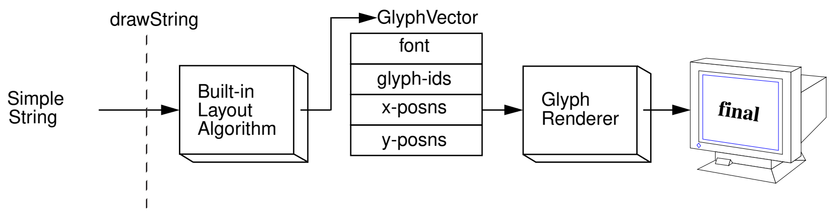


Figure 4-13 Using the Built-in Layout Algorithm

Figure 4-14 shows the process for using a custom layout algorithm. To use a custom layout algorithm, you must assemble all of the information necessary to lay out the text. The basic process is the same:

- Using the `Font`, determine which glyphs to use
- Determine where to place the glyphs
- Store this layout information in a `GlyphVector`

To render the text, you pass the `GlyphVector` to `drawString`, which in turn passes it to the glyph renderer. In Figure 4-14, the custom layout algorithm replaces the `fi` substring with the ligature *ﬁ*.

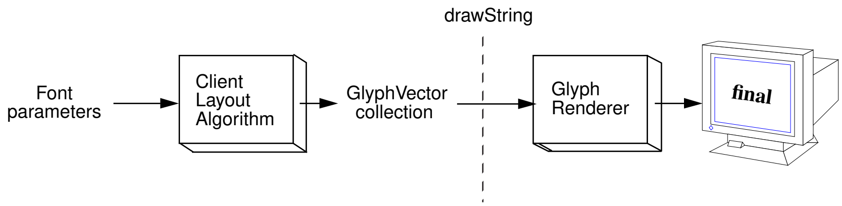


Figure 4-14 Using a Custom Layout Algorithm

4.6 Creating Font Derivations

Using the `Font.deriveFont` methods, you can create a new `Font` object with different attributes from an existing `Font` object. Often, a transform is applied to the existing `Font` to create a new derived `Font`. To do this, you:

1. Create a `Font` object.,
2. Create the `AffineTransform` you want to apply to the `Font`.
3. Call `Font.deriveFont`, passing in the `AffineTransform`.

In this way, you could easily create a `Font` in a custom size or a skewed version of an existing `Font`.

In the following code excerpt, an `AffineTransform` is applied to create a skewed version of the font Helvetica. The new derived font is then used to render a string.

```
// Create a transformation for the font.
AffineTransform fontAT = new AffineTransform();
fontAT.setToShear(-1.2, 0.0);
// Create a Font Object.
Font theFont = new Font("Helvetica", Font.PLAIN, 1);
// Derive a new font using the shear transform
theDerivedFont = theFont.deriveFont(fontAT);
// Add the derived font to the Graphics2D context
g2.setFont(theDerivedFont);
```

```
// Render a string using the derived font
g2.drawString("Java", 0.0f, 0.0f);
```

5

Imaging

The Java 2D™ API supports three imaging models

- The producer/consumer (push) model provided in previous versions of the JDK software.
- The immediate mode model introduced in the Java™ 2 SDK software release.
- The pipeline (pull) model compatible with the immediate mode model and that will be fully implemented in the forthcoming Java Advanced Imaging API.

The following table contrasts the features of each of these imaging models.

	Push Model	Immediate Mode Image Buffer Model	Pull Model
Major Interfaces/ Classes	• Image • ImageProducer • ImageConsumer • ImageObserver (JDK 1.0.x, 1.1.x)	• BufferedImage • Raster • BufferedImageOp • RasterOp (Java™ 2D API)	• RenderableImage • RenderableImageOp (Java 2D API) • RenderedOp • RenderableOp • Tiled Image (Java Advanced Imaging API)
Pros	• Processing driven by image availability (e.g. over network) • Images processed incrementally	• Simplest programming interface • Commonly used model	• Stores/processes only required data • Allows lazy evaluation
Cons	• Requires transfer (but not processing) of complete images • More complex programming interface	• Requires memory allocation of complete images • Requires processing of complete images	• More complex programming interface • More complex implementation

This chapter focuses on the objects and techniques of the immediate mode imaging model. The immediate mode imaging classes and interfaces of the Java 2D API provide techniques for dealing with pixel mapped images whose data is stored in memory. This API supports accessing image data in a variety of storage formats and manipulating image data through several types of filtering operations.

5.1 Interfaces and Classes

The immediate mode imaging APIs in the Java 2D™ API can be grouped into six categories: interfaces, image data classes, image operation classes, sample model classes, color model classes, and exceptions.

5.1.1 Imaging Interfaces

Interface	Description
`BufferedImageOp`	Describes single-input/single-output operations performed on `BufferedImage` objects. Implemented by `AffineTransformOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, and `RescaleOp`.
`RasterOp`	Defines single-input/single-output operations performed on `Raster` objects. Implemented by `AffineTransformOp`, `BandCombineOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, and `RescaleOp`.
`RenderedImage`	Defines a common protocol for objects that contain or can produce image data in the form of `Rasters`.
`WritableRenderedImage`	Extends: `RenderedImage` Defines a common protocol for objects that contain or can produce image data in the form of `Rasters` which can be modified.
`TileObserver`	Defines a protocol for objects that want to be notified when the modification state of a `WritableRenderedImage` changes.

5.1.2 Image Data Classes

Class	Description
`BufferedImage`	Extends: `Image` Implements: `WriteableRenderedImage` An image with an accessible data buffer. A `BufferedImage` has a `ColorModel` and a `Raster` of image data.
`ByteLookupTable`	Extends: `LookupTable` A `LookupTable` that contains byte data.
`DataBuffer`	Wraps one or more data arrays holding pixel data. Each data array is called a *bank*.

`DataBufferByte`	Extends: `DataBuffer` (Final) A data buffer that stores bytes of data. (Used in Java Advanced Imaging API)
`DataBufferInt`	Extends: `DataBuffer` (Final)) A data buffer that stores integer data.(Used in Java Advanced Imaging API)
`DataBufferShort`	Extends: `DataBuffer` (Final) A data buffer that stores short data.(Used in Java Advanced Imaging API)
`DataBufferUShort`	Extends: `DataBuffer` (Final) A data buffer that stores unsigned short data.
`Kernel`	A matrix that describes how an input pixel and its surrounding pixels affect the value of an output pixel in a `ConvolveOp` filtering operation.
`LookupTable`	Extends: `Object` A table that maps values from single-banded pixel data to color values.
`Raster`	A rectangular array of pixels from which you can retrieve image data. A `Raster` contains a `DataBuffer` and a `SampleModel`.
`ShortLookupTable`	Extends: `LookupTable` A lookup table that contains short data.
`WritableRaster`	Extends: `Raster` A `Raster` that you can modify.

5.1.3 Image Operation Classes

Class	**Description**
`AffineTransformOp`	Implements: `BufferedImageOp`, `RasterOp` A class that defines an affine transform to perform a linear mapping from 2D coordinates in a source `Image` or `Raster` to 2D coordinates in the destination image or `Raster`. This class can perform either bilinear or nearest neighbor affine transform operations.
`BandCombineOp`	Implements: `RasterOp` Using a specified matrix, this operation performs an arbitrary linear combination of bands in a `Raster`.
`BufferedImageFilter`	Extends: `ImageFilter` An `ImageFilter` that provides a simple means of using a `BufferedImageOp` (a single-source/single-destination image operator) to filter a `BufferedImage` or `Raster`.

Class	Description
ColorConvertOp	Implements: BufferedImageOp, RasterOp Performs a pixel-by-pixel color conversion of the data in the source image.
ConvolveOp	Implements: BufferedImageOp, RasterOp Uses a Kernel to perform a convolution on the source image. A convolution is a spatial operation where the pixels surrounding the input pixel are multiplied by a kernel value to generate the value of the output pixel. The Kernel mathematically defines the relationship between the pixels in the immediate neighborhood of the input pixel and the output pixel.
LookupOp	Implements: BufferedImageOp, RasterOp Performs a lookup operation from the source to the destination. For Rasters, the lookup operates on sample values. For BufferedImages, the lookup operates on color and alpha components.
RescaleOp	Implements: BufferedImageOp, RasterOp Performs a pixel-by-pixel rescaling of the data in the source image by multiplying each pixel value by a scale factor and then adding an offset.

5.1.4 Sample Model Classes

Class	Description
BandedSampleModel	Extends: ComponentSampleModel (Final) Provides access to image data stored with like samples stored as bands in separate banks of a DataBuffer. A pixel consists of one sample from each band.
ComponentSampleModel	Extends: SampleModel Provides access to image data stored with each sample of a pixel residing in a separate element of a DataBuffer. Different types of pixel interleaving are supported.
MultiPixelPackedSampleModel	Extends: SampleModel Provides access to image data stored with multiple one-sample pixels packed into one element of a DataBuffer.
PixelInterleavedSampleModel	Extends: ComponentSampleModel Provides access to image data stored with the sample data for each pixel in adjacent elements of the data array, and all elements in a single bank of a DataBuffer.

Class	Description
`SampleModel`	An abstract class that defines a mechanism for extracting sample data from an image without knowing how the underlying data is stored in a `DataBuffer`.
`SinglePixelPackedSampleModel`	Extends: `SampleModel` Provides access to image data stored with all the samples belonging to an individual pixel packed into one element of a `DataBuffer`.

5.1.5 Color Model Classes

Class	Description
`ColorModel`	Implements: `Transparency` JDK1.1 class. An abstract class that defines methods for translating from image pixel values to color components such as red, green, and blue.
`ComponentColorModel`	Extends: `ColorModel` A `ColorModel` that can handle an arbitrary `ColorSpace` and an array of color components to match the `ColorSpace`. This class can be used to represent most color models on most types of `GraphicsDevices`.
`DirectColorModel`	Extends: `PackedColorModel` JDK1.1 class. A `ColorModel` that represents pixel values that have RGB color components embedded directly in the bits of the pixel. This color model is similar to an X11 TrueColor visual. The default RGB `ColorModel` returned by `ColorModel.getRGBdefault` is a `DirectColorModel`.
`IndexColorModel`	Extends: `ColorModel` JDK1.1 class. A `ColorModel` that represents pixel values that are indices into a fixed color map in the sRGB `ColorSpace`.
`PackedColorModel`	Extends: `ColorModel` An abstract `ColorModel` that represents pixel values that have color components embedded directly in the bits of a pixel. `DirectColorModel` extends `PackedColorModel` to support pixels that contain RGB color components.

5.1.6 Exception Classes

Class	Description
`ImagingOpException`	Extends: `RuntimeException` Thrown if one of the BufferedImageOp or RasterOp filter methods can't process the image.
`RasterFormatException`	Extends: `RuntimeException` Thrown if there is invalid layout information in the Raster.

5.2 Immediate Mode Imaging Concepts

The immediate mode imaging model supports fixed-resolution images stored in memory. The model also supports filtering operations on image data. A number of classes and interfaces are used in this model.

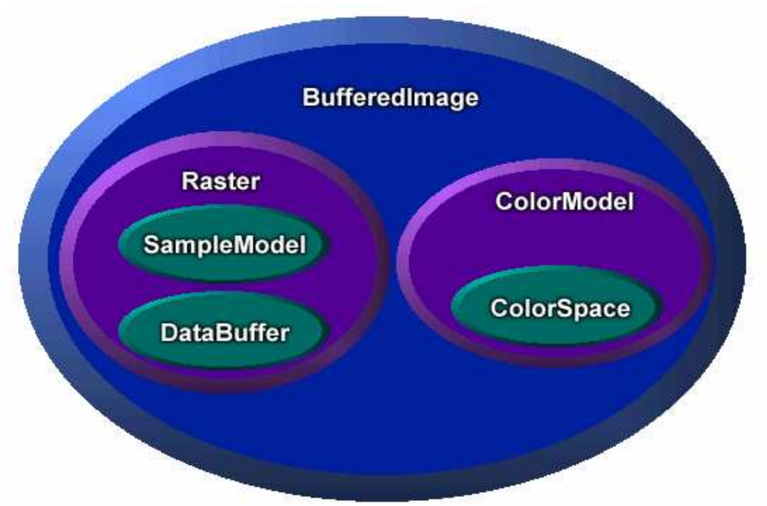


Figure 5-1 `BufferedImage` and supporting classes

As shown in Figure 5-1, `BufferedImage` provides general image management. A `BufferedImage` can be created directly in memory and used to hold and manipulate image data retrieved from a file or URL. A `BufferedImage` can be displayed using any `Graphics2D` object for a screen device, or rendered to any other destination using appropriate `Graphics2D` context. A `BufferedImage` object contains two other objects: a `Raster` and a `ColorModel`.

The `Raster` class provides image data management. It represents the rectangular coordinates of the image, maintains image data in memory, and provides a mechanism for creating multiple subimages from a single image data buffer. It also provides methods for accessing specific pixels within an image. A Raster object contains two other objects, a `DataBuffer` and a `SampleModel`.

The `DataBuffer` class holds pixel data in memory.

The `SampleModel` class interprets data in the buffer and provides it as individual pixels or rectangular ranges of pixels.

The `ColorModel` class provides a color interpretation of pixel data provided by the image's sample model.

The image package provides additional classes that define filtering operations on `BufferedImage` and `Raster` objects. Each image processing operation is embodied in a class that implements the `BufferedImageOp` interface, the `RasterOp` interface, or both interfaces. The operation class defines `filter` methods that performs the actual image manipulation.

Figure 5-2 illustrates the basic model for Java 2D™ API image processing:

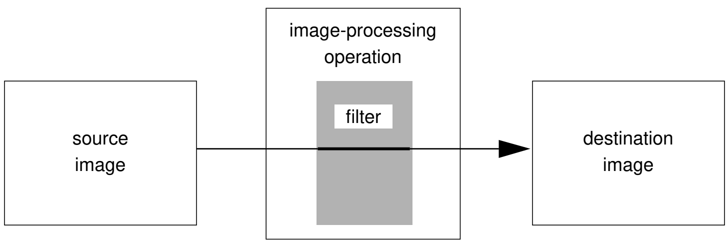


Figure 5-2 Image Processing Model

The operations supported include:

- Affine transformation
- Amplitude scaling
- Lookup-table modification
- Linear combination of bands
- Color conversion

- Convolution

Note that if you're interested just in displaying and manipulating images, you only need to understand the `BufferedImage` class and the filtering operation classes. On the other hand, if you're planning to write filters or otherwise directly access image data, you'll need to understand the classes associated with `BufferedImage`.

5.2.1 Terminology

Here are some terms used throughout the following discussions:

Data Elements: primitive types used as units of storage of image data. Data elements are individual members of a `DataBuffer` array. The layout of elements in the data buffer is independent of the interpretation of the data as pixels by an image's `SampleModel`.

Samples: distinct members of the pixels of an image. A `SampleModel` provides a mechanism for converting elements in the `DataBuffer` to pixels and their samples. The samples of a pixel may represent primary values in a particular color model. For example, a pixel in an RGB color model consists of three samples: red, green, and blue.

Components: values of pixels independent of color interpretation. The distinction between component and sample is useful with `IndexColorModel`, where pixel components are indexes into the `LookupTable`.

Band: the set of all samples of one type in an image, such as all red samples or all green samples. Pixel data can be stored in a number of ways, the two supported in the Java 2D API being banded and pixel interleaved. Banded storage organizes image data by bands, and a pixel is made up of sample data from the same position in each band. Pixel interleaved storage organizes image data by pixels, with a single array containing all pixels, and bands consisting of the set of samples at the same index position in each pixel.

Primaries: distinct members of a color value in a specific color model; for example the RGB model forms color values from the primaries red, green, and blue.

5.3 Using BufferedImages

The `BufferedImage` class is the main class supporting the immediate imaging mode. It manages an image in memory, providing ways to store pixel data, interpret pixel data, and to render the pixel data to a `Graphics` or `Graphics2D` context.

5.3.1 Creating a BufferedImage

To create a `BufferedImage`, call the `Component.createImage` method; this returns a `BufferedImage` whose drawing characteristics match those of the component used to create it—the created image is opaque, has the foreground and background colors of the `Component`, and you can't adjust the transparency of the image. You could use this technique when you want to do double buffered drawing for animation in a component; the discussion "Drawing in an Offscreen Buffer" on page 78 gives more details.

```
public Graphics2D createDemoGraphics2D(Graphics g) {
    Graphics2D g2 = null;
    int width = getSize().width;
    int height = getSize().height;

    if (offImg == null || offImg.getWidth() != width ||
                    offImg.getHeight() != height) {
       offImg = (BufferedImage) createImage(width, height);
    }

    if (offImg != null) {
        g2 = offImg.createGraphics();
        g2.setBackground(getBackground());
    }

    // .. clear canvas ..
    g2.clearRect(0, 0, width, height);

    return g2;
}
```

You can also create a blank `BufferedImage` in memory using one of several constructor methods provided.

5.3.2 Drawing in an Offscreen Buffer

The `BufferedImage` class can be used to prepare graphic elements offscreen then copy them to the screen. This technique is especially useful when a graphic is complex or used repeatedly. For example, if you want to display a complicated shape several times, you could draw it once into an offscreen buffer and then copy it to different locations in the window. By drawing the shape once and copying it, you can display the graphics more quickly.

The `java.awt` package facilitates the use of offscreen buffers by letting you draw to an `Image` object the same way that you draw to a window. All of the Java 2D™ API rendering features can be used when drawing to offscreen images.

Offscreen buffers are often used for animation. For example, you could use an offscreen buffer to draw an object once and then move it around in a window. Similarly, you could use an offscreen buffer to provide feedback as a user moves a graphic using the mouse. Instead of redrawing the graphic at every mouse location, you could draw the graphic once to an offscreen buffer, and then copy it to the mouse location as the user drags the mouse.[1]

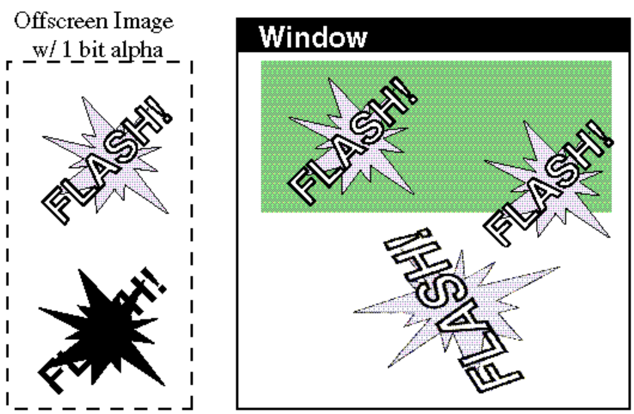


Figure 5-3 Using an Offscreen Buffer

Figure 5-3 demonstrates how a program can draw to an offscreen image and then copy that image into a window multiple times. The last time the image is copied, it is transformed. Note that transforming the image instead of redrawing it with the transformation might produce unsatisfactory results.

5.3.2.1 Creating an Offscreen Buffer

The simplest way to create an image that you can use as an offscreen buffer is to use the `Component.createImage` method.

1. It is up to the programmer to “erase” the previous version of the image before making a new copy at a new location. This can be done by redrawing the background or copying the background from another offscreen buffer.

By creating an image whose color space, depth, and pixel layout exactly match the window into which you are drawing, the image can be efficiently blitted to a graphics device. This allows `drawImage` to do its job quickly.

You can also construct a `BufferedImage` object directly to use as an offscreen buffer. This is useful when you need control over the offscreen image's type or transparency.

`BufferedImage` supports several predefined image types:

- `TYPE_3BYTE_BGR`
- `TYPE_4BYTE_ABGR`
- `TYPE_4BYTE_ABGR_PRE`
- `TYPE_BYTE_BINARY`
- `TYPE_BYTE_GRAY`
- `TYPE_BYTE_INDEXED`
- `TYPE_CUSTOM`
- `TYPE_INT_ARGB_PRE`
- `TYPE_INT_ARGB`
- `TYPE_INT_BGR`
- `TYPE_INT_RGB`
- `TYPE_USHORT_555_RGB`
- `TYPE_USHORT_565_RGB`
- `TYPE_INT_GRAY`

A `BufferedImage` object can contain an alpha channel. In Figure 5-3, an alpha channel is used to distinguish painted and unpainted areas, allowing an irregular shape to appear over graphics that have already been painted (in this case, a shaded rectangle). In other cases, you might use alpha channel to blend the colors of the new image into those in the existing image.

Note: unless you need alpha image data for transparency, as with the irregularly shaped images shown in Figure 5-2, you should avoid creating an off-screen buffer with alpha. Using alpha where it's unnecessary slows rendering performance.

`GraphicsConfiguration` provides convenience methods that automatically create buffered images in a format compatible with your configuration. You can also query the graphics configuration associated with the graphics device on which the window resides to get the information you need to construct a compatible `BufferedImage` object.

5.3.2.2 Drawing in an Offscreen Buffer

To draw in a buffered image, you call its `BufferedImage.createGraphics` method, which returns a `Graphics2D` object. With this object, you can call all of the `Graphics2D` methods to draw graphics primitives, place text, and render other images in the image. This drawing technique supports dithering and other enhancements provided by the 2D imaging package. The following code illustrates the use of offscreen buffering:

```
public void update(Graphics g){
    Graphics2D g2 = (Graphics2D)g;
    if(firstTime){
        Dimension dim = getSize();
        int w = dim.width;
        int h = dim.height;
        area = new Rectangle(dim);
        bi = (BufferedImage)createImage(w, h);
        big = bi.createGraphics();
        rect.setLocation(w/2-50, h/2-25);
        big.setStroke(new BasicStroke(8.0f));
        firstTime = false;
    }

    // Clears the rectangle that was previously drawn.
    big.setColor(Color.white);
    big.clearRect(0, 0, area.width, area.height);

    // Draws and fills the newly positioned rectangle to the buffer.
    big.setPaint(strokePolka);
    big.draw(rect);
    big.setPaint(fillPolka);
    big.fill(rect);

    // Draws the buffered image to the screen.
    g2.drawImage(bi, 0, 0, this);

}
```

5.3.3 Manipulating BufferedImage Data Directly

In addition to drawing directly in a `BufferedImage`, you can directly access and manipulate the image's pixel data in a couple of ways. These are useful if you're implementing the `BufferedImageOp` filtering interface, as described in "Image Processing and Enhancement" on page 83.

You can use the `BufferedImage.setRGB` methods to directly set the value of a pixel or a pixel array to a specific RGB value. Note that no dithering is performed when you modify pixels directly. You can also manipulate pixel data by manipulating a WritableRaster object associated with a BufferedImage (see“Managing and Manipulating Rasters” on page 79).

5.3.4 Filtering a BufferedImage

You can apply a filtering operation to a BufferedImage using an object that implements BufferedImageOp interface. Filtering and the classes that provide this filtering interface are discussed in “Image Processing and Enhancement” on page 83.

5.3.5 Rendering a BufferedImage

To render a buffered image into a specific context, call one of the drawImage method of the context’s Graphics object. For example, when rendering within a Component.paint method, you call drawImage on the graphics object passed to the method.

```
public void paint(Graphics g) {

    if (getSize().width <= 0 || getSize().height <= 0)
        return;

    Graphics2D g2 = (Graphics2D) g;

    if (offImg != null && isShowing())  {
        g2.drawImage(offImg, 0, 0, this);
    }
}
```

5.4 Managing and Manipulating Rasters

A BufferedImage object uses a Raster to manage its rectangular array of pixel data. The Raster class defines fields for the image’s coordinate system—width, height, and origin. A Raster object itself uses two objects to manage the pixel data, a DataBuffer and a SampleModel. The DataBuffer is the object that stores pixel data for the raster (as described on page 81), and the SampleModel provides the interpretation of pixel data from the DataBuffer (as described on page 81).

5.4.1 Creating a Raster

In most cases, you don't need to create a `Raster` directly, since one is supplied with any `BufferedImage` that you create in memory. However, one of the `BufferedImage` constructor methods allows you to create a `Raster` by passing in a `WritableRaster`.

The `Raster` class provides a number of static factory methods for creating `Rasters` with the `DataBuffers` and `SampleModels` you specify. You can use these factories when implementing `RasterOp` filtering classes.

5.4.2 Parent and Child Rasters

The `Raster` class incorporates the concept of parent and child rasters. This can improve storage efficiency by allowing you to construct any number of buffered images from the same parent. The parent and its children all refer to the same data buffer, and each child has a specific offset and bounds to identify its image location in the buffer. A child identifies its ownership through its `getParent` method.

To create a subraster, you use the `Raster.createSubRaster` method.When you create a subraster, you identify the area of its parent that it covers and its offset from the parent's origin.

5.4.3 Operations on a Raster

The `Raster` class defines a number of ways to access pixels and pixel data. These are useful when you're implementing the `RasterOp` interface, which provides raster-level filtering and manipulation of image data, or when implementing any method that needs to perform low-level pixel manipulation.

The `Raster.getPixel` methods let you get an individual pixel, which is returned as individual samples in an array. The `Raster.getDataElements` methods return a specified run of uninterpreted image data from the `DataBuffer`. The `Raster.getSample` method returns samples of an individual pixel. The `getSamples` method returns a band for a particular region of an image.

In addition to these methods, you can also access the data buffer and the sample model through instance variables of the `Raster` class. These objects provide additional ways to access and interpret the `Raster`'s pixel data.

5.4.4 The WritableRaster Subclass

The `WritableRaster` subclass provides methods for setting pixel data and samples. The `Raster` associated with a `BufferedImage` is actually a `WritableRaster`, thus providing full access to manipulate its pixel data.

5.5 Image Data and DataBuffers

The `DataBuffer` belonging to a `Raster` represents an array of image data. When you create a `Raster` directly or through the `BufferedImage` constructors, you specify a width and height in pixels, along with a `SampleModel` for the image data. This information is used to create a `DataBuffer` of the appropriate data type and size.

There are three subclasses of `DataBuffer`, each representing a different type of data element:

- `DataBufferByte` (represents 8-bit values)
- `DataBufferInt` (represents 32-bit values)
- `DataBufferShort` (represents 16-bit values)
- `DataBufferUShort` (represents unsigned short values)

As defined earlier, elements are the discrete members of the array of the data buffer, and components or samples are the discrete values that together make up a pixel. There can be various mappings between a particular type of element in a `DataBuffer` and a particular type of pixel represented by a `SampleModel`. It is the responsibility of the various `SampleModel` subclasses to implement that mapping and provide a way to get specific pixels from a specific `DataBuffer`.

`DataBuffer` constructors provide ways to create buffers of a specific size and a specific number of banks.

While you can access image data in a `DataBuffer` directly, it's generally easier and more convenient to do so through the methods of the `Raster` and `WritableRaster` classes.

5.6 Extracting Pixel Data from a SampleModel

The abstract `SampleModel` class defines methods for extracting samples of an image without knowing how the underlying data is stored. The class provides fields for tracking the height and width of the image data in the associated

`DataBuffer`, and for describing the number of bands and the data type of that buffer. `SampleModel` methods provide image data as a collection of pixels, with each pixel consisting of a number of samples or components.

The `java.awt.image` package provides five types of sample models:

- `ComponentSampleModel`—used to extract pixels from images that store sample data in separate data array elements in one bank of a `DataBuffer`.
- `BandedSampleModel`—used to extract pixels from images that store each sample in a separate data element with bands stored in a sequence of data elements
- `PixelInterleavedSampleModel`—used to extract pixels from images that store each sample in a separate data element with pixels stored in a sequence of data elements.
- `MultiPixelPackedSampleModel`—used to extract pixels from single banded images that store multiple one-sample pixels in one data element.
- `SinglePixelPackedSampleModel`—used to extract samples from images that store sample data for a single pixel in one data array element in the first bank of a `DataBuffer`.

Pixel data presented by the `SampleModel` may or may not correlate directly to a color data representation of a particular color model, depending on the data source. For example, in photographic image data, the samples may represent RGB data. In image data from a medical imaging device, samples can represent different types of data such as temperature or bone density.

There are three categories of methods for accessing image data. The `getPixel` methods return a whole pixel as an array, with one entry for each sample. The `getDataElement` methods provide access to the raw, uninterpreted data stored in the `DataBuffer`. The `getSample` methods provide access to pixel components for a specific band.

5.7 ColorModels and Color Data

In addition to the `Raster` object for managing image data, the `BufferedImage` class includes a `ColorModel` for interpreting that data as color pixel values. The abstract `ColorModel` class defines methods for turning an image's pixel data into a color value in its associated `ColorSpace`.

The `java.awt.image` package provides four types of color models:

- `PackedColorModel`—An abstract `ColorModel` that represents pixel values that have color components embedded directly in the bits of an integer pixel. A `DirectColorModel` is a subclass of `PackedColorModel`.
- `DirectColorModel`—a `ColorModel` that represents pixel values that have RGB color components embedded directly in the bits of the pixel itself. `DirectColorModel` model is similar to an X11 TrueColor visual.
- `ComponentColorModel`—a `ColorModel` that can handle an arbitrary `ColorSpace` and an array of color components to match the `ColorSpace`.
- `IndexColorModel`—a `ColorModel` that represents pixel values that are indices into a fixed color map in the sRGB color space.

`ComponentColorModel` and `PackedColorModel` are new in the Java™ 2 SDK software release.

Based on data in the `DataBuffer`, the `SampleModel` provides the `ColorModel` with a pixel, which the `ColorModel` then interprets as a color.

5.7.1 Lookup Table

A lookup table contains data for one or more channels or image components; for example, separate arrays for R, G, and B. The `java.awt.image` package defines two types of lookup tables that extend the abstract `LookupTable` class, one that contains byte data and one that contains short data (`ByteLookupTable` and `ShortLookupData`).

5.8 Image Processing and Enhancement

The image package provides a pair of interfaces that define operations on `BufferedImage` and `Raster` objects: `BufferedImageOp` and `RasterOp`.

The classes that implement these interfaces include `AffineTransformOp`, `BandCombineOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, `RescaleOp`. These classes can be used to geometrically transform, blur, sharpen, enhance contrast, threshold, and color correct images.

Figure 5-4 illustrates edge detection and enhancement, an operation that emphasizes sharp changes in intensity within an image. Edge detection is commonly used in medical imaging and mapping applications. Edge detection is used to increase the contrast between adjacent structures in an image, allowing the viewer to discriminate greater detail.

Figure 5-4 Edge detection and enhancement

The following code illustrates edge detection:

```
float[] elements = { 0.0f, -1.0f, 0.0f,
                    -1.0f, 4.f, -1.0f,
                    0.0f, -1.0f, 0.0f};
...

BufferedImage bimg = new
BufferedImage(bw,bh,BufferedImage.TYPE_INT_RGB);
Kernel kernel = new Kernel(3, 3, elements);
ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP,
                                null);
cop.filter(bi,bimg);
```

Figure 5-5 demonstrates lookup table manipulation. A lookup operation can be used to alter individual components of a pixel.

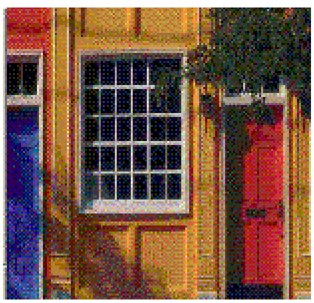


Figure 5-5 Lookup-table Manipulation

The following code demonstrates Lookup-table manipulation:

```
byte reverse[] = new byte[256];
   for (int j=0; j<200; j++){
       reverse[j]=(byte)(256-j);
   }
   ByteLookupTable blut=new ByteLookupTable(0, reverse);
   LookupOp lop = new LookupOp(blut, null);
   lop.filter(bi,bimg);
```

Figure 5-6 illustrates rescaling. Rescaling can increase or decrease the intensity of all points. Rescaling can be used to increase the dynamic range of an otherwise neutral image, bringing out detail in a region that appears neutral or flat.

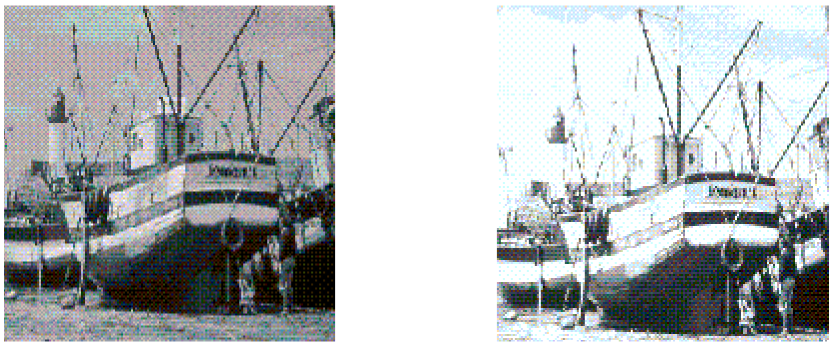


Figure 5-6 Rescaling

The following code snippet illustrates rescaling:

```
RescaleOp rop = new RescaleOp(1.5f, 1.0f, null);
rop.filter(bi,bimg);
```

5.8.1 Using an Image Processing Operation

Convolution is the process that underlies most spatial filtering algorithms. Convolution is the process of weighting or averaging the value of each pixel in an image with the values of neighboring pixels.This allows each output pixel to be affected by the immediate neighborhood in a way that can be mathematically specified with a kernel. Figure 5-7 illustrates Convolution.

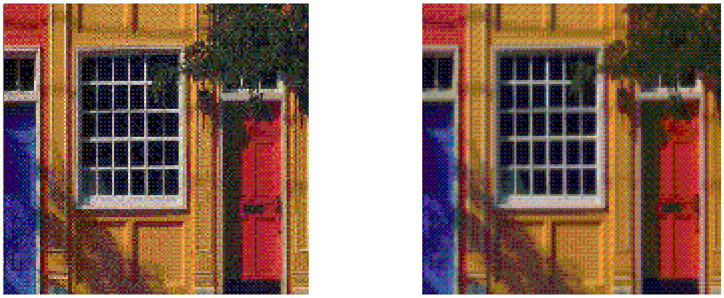


Figure 5-7 Blurring with Convolution

The following code fragment illustrates how to use one of the image processing classes, `ConvolveOp`. In this example, each pixel in the source image is averaged equally with the eight pixels that surround it.

```
float weight = 1.0f/9.0f;
float[] elements = new float[9]; // create 2D array

// fill the array with nine equal elements
for (i = 0; i < 9; i++) {
   elements[i] = weight;
}
// use the array of elements as argument to create a Kernel
private Kernel myKernel = new Kernel(3, 3, elements);
public ConvolveOp simpleBlur = new ConvolveOp(myKernel);

// sourceImage and destImage are instances of BufferedImage
simpleBlur.filter(sourceImage, destImage) // blur the image
```

The variable `simpleBlur` contains a new instance of `ConvolveOp` that implements a blur operation on a `BufferedImage` or a `Raster`. Suppose that `sourceImage` and `destImage` are two instances of `BufferedImage`. When you call `filter`, the core method of the `ConvolveOp` class, it sets the value of each pixel in the destination image by averaging the corresponding pixel in the source image with the eight pixels that surround it.

The convolution kernel in this example could be represented by the following matrix, with elements specified to four significant figures:

$$\mathbf{K} = \begin{bmatrix} 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \end{bmatrix}$$

When an image is convolved, the value of each pixel in the destination image is calculated by using the kernel as a set of weights to average the pixel's value with the values of surrounding pixels. This operation is performed on each channel of the image.

The following formula shows how the weights in the kernel are associated with the pixels in the source image when the convolution is performed. Each value in the kernel is tied to a spatial position in the image.

$$\mathbf{K} = \begin{bmatrix} i-1, j-1 & i, j-1 & i+1, j-1 \\ i-1, j & i, j & i+1, j \\ i-1, j+1 & i, j+1 & i+1, j+1 \end{bmatrix}$$

The value of a destination pixel is the sum of the products of the weights in the kernel multiplied by the value of the corresponding source pixel. For many simple operations, the kernel is a matrix that is square and symmetric, and the sum of its weights adds up to one.[2]

The convolution kernel in this example is relatively simple. It weights each pixel from the source image equally. By choosing a kernel that weights the source image at a higher or lower level, a program can increase or decrease the intensity of the destination image. The `Kernel` object, which is set in the `ConvolveOp` constructor, determines the type of filtering that is performed. By setting other values, you can perform other types of convolutions, including blurring (such as Gaussian blur, radial blur, and motion blur), sharpening, and smoothing operations. Figure 5-8 illustrates sharpening using Convolution.

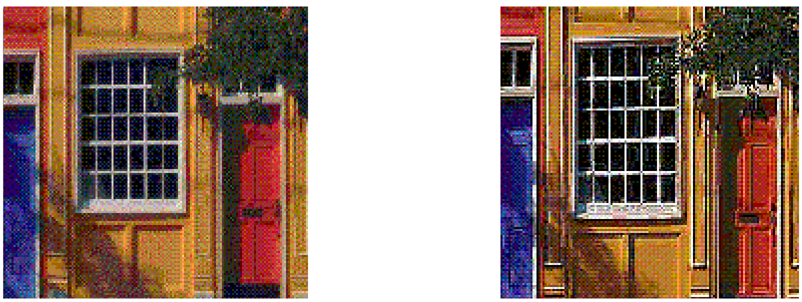


Figure 5-8 Sharpening with Convolution

The following code snippet illustrates sharpening with Convolution:

2. If the sum of the weights in the matrix is one, the intensity of the destination image is unchanged from the source.

```
float[] elements = { 0.0f, -1.0f, 0.0f,
                    -1.0f,  5.f, -1.0f,
                     0.0f, -1.0f,  0.0f};
...

Kernel kernel = new Kernel(3,3,elements);
ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP,
                                  null);
cop.filter(bi,bimg);
```

6

Color

Color imaging is one of the fundamental components of any graphics system, and it is often a source of great complexity in the imaging model. The Java 2D™ API provides support for high-quality color output that is easy to use and allows advanced clients to make sophisticated use of color.

The key color management classes in the Java 2D API are `ColorSpace`, `Color`, `ColorModel`:

- A `ColorSpace` represents a system for measuring colors, typically using three separate numerical values or components. The `ColorSpace` class contains methods for converting between the color space and two standard color spaces, `CIEXYZ` and RGB.
- A `Color` is a fixed color, defined in terms of its components in a particular `ColorSpace`. To draw a `Shape` in a color, such as red, you pass a `Color` object representing that color to the `Graphics2D` context. `Color` is defined in the `java.awt package`.
- A `ColorModel` describes a particular way that pixel values are mapped to colors. A `ColorModel` is typically associated with an `Image` or `BufferedImage` and provides the information necessary to correctly interpret the pixel values. `ColorModel` is defined in the `java.awt.image package`.

6.1 Classes

Class	Description
`ColorSpace`	Identifies the color space of a `Color` object, `Image`, `BufferedImage`, or `GraphicsDevice`. Has methods to transform between RGB and CIEXYZ color spaces.

Class	Description
`ICC_ColorSpace`	Extends: `ColorSpace` Represents device-independent and device-dependent color spaces based on the ICC Profile Format Specification.
`ICC_Profile`	A representation of color profile data for device independent and device dependent color spaces based on the ICC Profile Format Specification.
`ICC_ProfileGray`	Extends: `ICC_Profile` A representation of color space type gray.
`ICC_ProfileRGB`	Extends: `ICC_Profile` A representation of color space type RGB.

6.2 Color Concepts

A `ColorModel` is used to interpret pixel data in an image. This includes mapping components in the bands of an image to components of a particular color space. It might also involve extracting pixel components from packed pixel data, retrieving multiple components from a single band using masks, and converting pixel data through a lookup table.

To determine the color value of a particular pixel in an image, you need to know how color information is encoded in each pixel. The `ColorModel` associated with an image encapsulates the data and methods necessary for translating a pixel value to and from its constituent color components.

The Java 2D™ API provides two color models in addition to the `DirectColorModel` and `IndexColorModel` defined in the JDK 1.1 software release:

- `ComponentColorModel` can handle an arbitrary `ColorSpace` and an array of color components to match the `ColorSpace`. This model can be used to represent most color models on most types of `GraphicsDevices`.
- `PackedColorModel` is a base class for models that represent pixel values that have their color components embedded directly in the bits of an integer pixel. A `PackedColorModel` stores the packing information that describes how color and alpha components are extracted from the channel.The `DirectColorModel` in the JDK 1.1 software release is a `PackedColorModel`.

6.2.0.1 ColorSpace

A `ColorSpace` object represents a system for measuring colors, typically using three separate numeric values. For example, RGB and CMYK are color spaces. A

`ColorSpace` object serves as a colorspace tag that identifies the specific color space of a `Color` object or, through a `ColorModel` object, of an `Image`, `BufferedImage`, or `GraphicsConfiguration`. `ColorSpace` provides methods that transform `Colors` in a specific color space to and from `sRGB` and to and from a well-defined `CIEXYZ` color space.

All `ColorSpace` objects must be able to map a color from the represented color space into `sRGB` and transform an `sRGB` color into the represented color space. Since every `Color` contains a `ColorSpace` object, set explicitly or by default, every `Color` can also be converted to `sRGB`. Every `GraphicsConfiguration` is associated with a `ColorSpace` object that in turn has an associated `ColorSpace`. A color specified in any color space can be displayed by any device by mapping it through `sRGB` as an intermediate color space.

The methods used for this process are `toRGB` and `fromRGB`:

- `toRGB` transforms a `Color` in the represented color space to a `Color` in `sRGB`.
- `fromRGB` takes a `Color` in `sRGB` and transforms it into the represented color space.

Though mapping through `sRGB` always works, it's not always the best solution. For one thing, `sRGB` cannot represent every color in the full gamut of `CIEXYZ` colors. If a color is specified in some space that has a different gamut (spectrum of representable colors) than `sRGB`, then using `sRGB` as an intermediate space results in a loss of information. To address this problem, the `ColorSpace` class can map colors to and from another color space, the “conversion space” `CIEXYZ`.

The methods `toCIEXYZ` and `fromCIEXYZ` map color values from the represented color space to the conversion space. These methods support conversions between any two color spaces at a reasonably high degree of accuracy, one `Color` at a time. However, it is expected that Java 2D API implementations will support high-performance conversion based on underlying platform color-management systems, operating on entire images. (See `ColorConvertOp` in “Imaging” on page 67.)

Figure 6-1 and Figure 6-2 illustrate the process of translating a color specified in a CMYK color space for display on an RGB color monitor. Figure 6-1 shows a mapping through `sRGB`. As this figure illustrates, the translation of the CMYK color to an RGB color is not exact because of a gamut mismatch. [1]

1. Of course, the colors used in these diagrams are illustrative, not accurate. The point is that colors might not be mapped accurately between color spaces unless an appropriate conversion space is used.

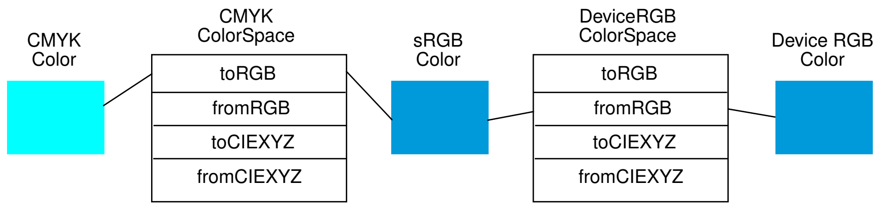


Figure 6-1 Mapping Through sRGB

Figure 6-2 shows the same process using `CIEXYZ` as the conversion space. When `CIEXYZ` is used, the color is passed through accurately.

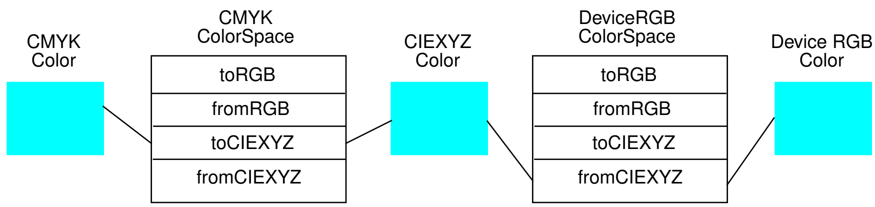


Figure 6-2 Mapping Through CIEXYZ

6.2.0.2 ICC_Profile and ICC_ColorSpace

`ColorSpace` is actually an abstract class. The Java 2D API provides one implementation, `ICC_ColorSpace`, which is based on ICC Profile data as represented by the `ICC_Profile` class. You can define your own subclasses to represent arbitrary color spaces, as long as the methods discussed above are implemented. However, most developers can simply use the default `sRGB ColorSpace` or color spaces that are represented by commonly available ICC Profiles, such as profiles for monitors and printers, or profiles embedded in image data.

"ColorSpace" on page 90 describes how `ColorSpace` objects represent a color space and how colors in the represented space can be mapped to and from a conversion space. Color management systems are often used to handle the mapping between color spaces. A typical color management system (CMS) manages ICC profiles, which are similar to `ColorSpace` objects; ICC profiles describe an input space and a connection space, and define how to map between them. Color management systems are very good at figuring out how to map a color tagged with one profile into the color space of another profile.

The Java 2D API defines a class called ICC_Profile that holds data for an arbitrary ICC Profile. ICC_ColorSpace is an implementation of the abstract ColorSpace class. ICC_ColorSpace objects can be constructed from ICC_Profiles. (There are some limitations—not all ICC Profiles are appropriate for defining an ICC_ColorSpace).

ICC_Profile has several subclasses that correspond to specific color space types, such as ICC_ProfileRGB and ICC_ProfileGray. Each subclass of ICC_Profile has a well-defined input space (such as an RGB space) and a well-defined connection space (like CIEXYZ). The Java 2D API can use a platform's CMS to access color profiles for various devices such as scanners, printers, and monitors. It can also use the CMS to find the best mapping between profiles.

6.2.1 Describing Colors

The Color class provides a description of a color in a particular color space. An instance of Color contains the value of the color components and a ColorSpace object. Because a ColorSpace object can be specified in addition to the color components when a new instance of Color is created, the Color class can handle colors in any color space.

The Color class has a number of methods that support a proposed standard RGB color space called sRGB (see http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html). sRGB is the default color space for the Java 2D API. Several constructors defined by the Color class omit the ColorSpace parameter. These constructors assume that the color's RGB values are defined in sRGB, and use a default instance of ColorSpace to represent that space.

The Java 2D API uses sRGB as a convenience to application programmers, not as a reference color space for color conversion. Many applications are primarily concerned with RGB images and monitors, and defining a standard RGB color space makes writing such applications easier. The ColorSpace class defines the methods toRGB and fromRGB so that developers can easily retrieve colors in this standard space. These methods are not intended to be used for highly accurate color correction or conversions. See “ColorSpace” on page 90 for more information.

To create a color in a color space other than sRGB, you use the Color constructor that takes a ColorSpace object and an array of floats that represent the color components appropriate to that space. The ColorSpace object identifies the color space.

To display a rectangle of a certain color, such as the process color cyan, you need a way to describe this color to the system. There are a number of different ways to describe a color; for example, a color could be described as a set of red, green, and blue (RGB) components, or a set of cyan, magenta, yellow, and black (CMYK) components. These different techniques for specifying colors are called *color spaces*.

As you probably know, colors on a computer screen are generated by blending different amounts of red, green, and blue light. Therefore, using an RGB color space is standard for imaging on computer monitors. Similarly, four-color process printing uses cyan, magenta, yellow, and black ink to produce color on a printed page; the printed colors are specified as percentages in a CMYK color space.

Due to the prevalence of computer monitors and color printing, RGB and CMYK color spaces are both commonly used to describe colors. However, both types of color spaces have a fundamental drawback—they are device-dependent. The cyan ink used by one printer might not exactly match the cyan ink used by another. Similarly, a color described as an RGB color might look blue on one monitor and purplish on another.

6.2.2 Mapping Colors through sRGB and CIEXYZ

The Java 2D API refers to RGB and CMYK as color space types. A particular model of monitor with its particular phosphors defines its own RGB color space. Similarly, a particular model of printer has its own CMYK color space. Different RGB or CMYK color spaces can be related to each other through a device-independent color space.

Standards for the device-independent specification of color have been defined by the International Commission on Illumination (CIE). The most commonly used device-independent color space is the three-component XYZ color space developed by CIE. When you specify a color using `CIEXYZ`, you are insulated from device dependencies.

Unfortunately, it's not always practical to describe colors in the `CIEXYZ` color space—there are valid reasons for representing colors in other color spaces. To obtain consistent results when a color is represented using a device-dependent color space such as a particular RGB space, it is necessary to show how that RGB space relates to a device-independent space like `CIEXYZ`.

One way to map between color spaces is to attach information to the spaces that describes how the device-dependent space relates to the device-independent space. This additional information is called a *profile*. A commonly used type of color profile is the ICC Color Profile, as defined by the International Color Consortium. For details, see the ICC Profile Format Specification, version 3.4 available at `http://www.color.org`.

Figure 6-3 illustrates how a solid color and a scanned image are passed to the Java 2D API, and how they are displayed by various output devices. As you can see in Figure 6-3, both the input color and the image have profiles attached.

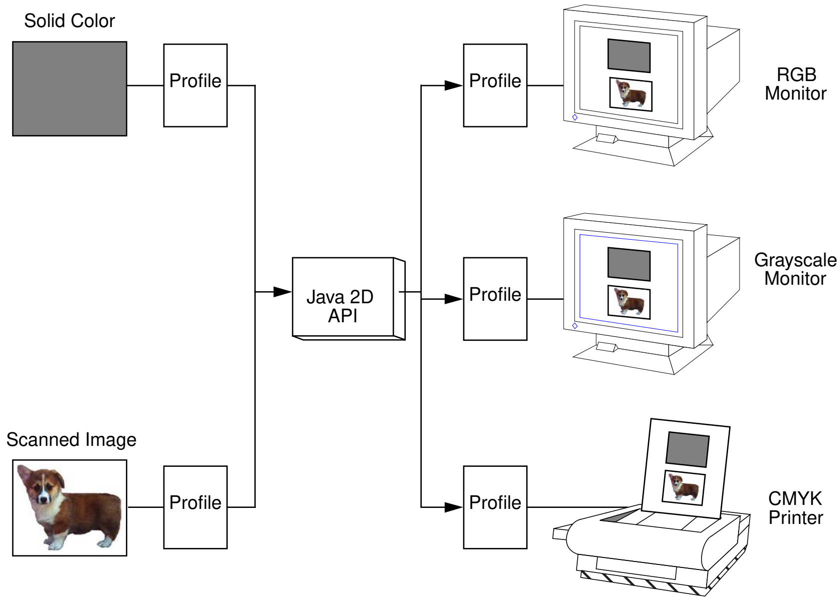


Figure 6-3 Using Profiles to Map Between Color Spaces

6.2.2.1 Color Matching

Once the API has an accurately specified color, it must reproduce that color on an output device, such as a monitor or printer. These devices have imaging characteristics of their own that must be taken into account to make sure that they produce the correct results. Another profile is associated with each output device to describe how the colors need to be transformed to produce accurate results.

Achieving consistent and accurate color requires that both input colors and output devices be profiled against a standard color space. For example, an input color could be mapped from its original color space into a standard device-independent space, and then mapped from that space to the output device's color space. In many respects, the transformation of colors mimics the transformation of graphical objects in an (x, y) coordinate space. In both cases, a transformation is used to specify coordinates in a "standard" space and then map those coordinates to a device-specific space for output.

7 Printing

The Java Printing API enables applications to:

- Print all AWT and Java 2D™ graphics, including composited graphics and images.
- Control document-composition functions such as soft collating, reverse order printing, and booklet printing.
- Invoke printer-specific functions such as duplex (two-sided) printing and stapling.
- Print on all platforms, including Windows and Solaris. This includes printers directly attached to the computer as well as those that the platform software is able to access using network printing protocols.

Not all of these features are supported in the Java™ 2 SDK Printing API and implementation. The API will be extended to support all of these features in future releases. For example, additional printer controls will be added by augmenting the set of named properties of a print job that the application can control.

7.1 Interfaces and Classes

Interface	Description
`Printable`	The `Printable` interface is implemented by each *page painter*, the application class(es) called by the printing system to render a page. The system calls the page painter's `print` method to request that a page be rendered.
`Pageable`	The `Pageable` interface is implemented by a document that is to be printed by the printing system. Through the `Pageable` methods, the system can determine the number of pages in the document, the format to use for each page, and the page painter to use to render each page.
`PrinterGraphics`	The `Graphics2D` objects that a page painter uses to render a page implement the `PrinterGraphics` interface. This enables an application to get the `PrinterJob` object that is controlling the printing.

Class	Description
`Book`	Implements: `Pageable` Represents a document in which pages can have different page formats and page painters. This class uses the `Pageable` interface to interact with a `PrinterJob`.
`PageFormat`	Describes the size and orientation of a page to be printed, as well as the `Paper` used to print it. For example, *portrait* and *landscape* paper orientations are represented by `PageFormat`.
`Paper`	Describes the physical characteristics of a piece of paper.
`PrinterJob`	The principal class that controls printing. The application calls `PrinterJob` methods to set up a job, display a print dialog to the user (optional), and to print the pages in the job.

7.2 Printing Concepts

The Java Printing API is based on a *callback* printing model in which the printing system, not the application, controls when pages are printed. The application provides information about the document to be printed and the printing system asks the application to render each page as it needs them.

The printing system might request that a particular page be rendered more than once or request that pages be rendered out of order. The application must be able to generate the proper page image, no matter which page the printing system requests. In this respect, the printing system is similar to the window toolkit, which can request components to repaint at any time, in any order.

The callback printing model is more flexible than traditional application-driven printing models and supports printing on a wider range of systems and printers. For example, if a printer stacks output pages in reverse order, the printing system can ask the application to generate pages in reverse order so that the final stack is in proper reading order.

This model also enables applications to print to a bitmap printer from computers that don't have enough memory or disk space to buffer a full-page bitmap. In this situation, a page is printed as a series of small bitmaps or *bands*. For example, if only enough memory to buffer one tenth of a page is available, the page is divided into ten bands. The printing system asks the application to render each page ten times, once to fill each band. The application does not need to be aware of the number or size of the bands; it simply must be able to render each page when requested.

7.2.1 Supporting Printing

An application has to perform two tasks to support printing:

- Job control—initiating and managing the print job.
- Imaging—rendering each page when the printing system requests it.

7.2.1.1 Job Control

The user often initiates printing by clicking a button or selecting a menu item in an application. When a print operation is triggered by the user, the application creates a `PrinterJob` object and uses it to manage the printing process.

The application is responsible for setting up the print job, displaying print dialogs to the user, and starting the printing process.

7.2.1.2 Imaging

When a document is printed, the application has to render each page when the printing system requests it. To support this mechanism, the application provides a *page painter* that implements the `Printable` interface. When the printing system needs a page rendered, it calls the page painter's `print` method.

When a page painter's `print` method is called, it is passed a `Graphics` context to use to render the page image. It is also passed a `PageFormat` object that specifies the geometric layout of the page, and an integer *page index* that identifies the ordinal position of the page in the print job.

The printing system supports both `Graphics` and `Graphics2D` rendering, To print Java 2D™ `Shapes`, `Text`, and `Images`, you cast the `Graphics` object passed into the `print` method to a `Graphics2D`.

To print documents in which the pages use different page painters and have different formats, you use a *pageable job*. To create a pageable job, you can use the `Book` class or your own implementation of the `Pageable` interface. To implement simple printing operations, you do not need to use a pageable print job; `Printable` can be used as long as all of the pages share the same page format and painter.

7.2.2 Page Painters

The principal job of a page painter is to render a page using the graphics context that is provided by the printing system. A page painter implements the `Printable.print` method:

```
public int print(Graphics g, PageFormat pf, int pageIndex)
```

The graphics context passed to the `print` method is either an instance of `Graphics` or `Graphics2D`, depending on the packages loaded in your Java Virtual Machine. To use `Graphics2D` features, you can cast the `Graphics` object to a `Graphics2D`. The `Graphics` instance passed to `print` also implements the `PrinterGraphics` interface.

The `PageFormat` passed to a `Printable` describes the geometry of the page being printed. The coordinate system of the graphics context passed to `print` is fixed to the page: the origin of the coordinate system is at the upper left corner of the paper, *X* increases to the right, *Y* increases downward, and the units are 1/72 inch. If the page is in portrait orientation, the x-axis aligns with the paper's "width," while the y-axis aligns with the paper's "height." (Normally, but not always, a paper's height exceeds its width.) If the page is in landscape orientation, the roles are reversed: the x-axis aligns with the paper's "height" and the y-axis with its "width."

Because many printers cannot print on the entire paper surface, the `PageFormat` specifies the *imageable area* of the page: this is the portion of the page in which it's safe to render. The specification of the imageable area does not alter the coordinate system; it is provided so that the contents of the page can be rendered so that they don't extend into the area where the printer can't print.

The graphics context passed to `print` has a clip region that describes the portion of the imageable area that should be drawn. It is always safe to draw the entire page into the context; the printing system will handle the necessary clipping.

However, to eliminate the overhead of drawing portions of the page that won't be printed, you can use the clipping region to limit the areas that you render. To get the clipping region from the graphics context, call `Graphics.getClip`. You are strongly encouraged to use the clip region to reduce the rendering overhead.

It is sometimes desirable to launch the entire printing operation "in the background" so that a user can continue to interact with the application while pages are being rendered. To do this, call `PrinterJob.print` in a separate thread.

If possible, you should avoid graphics operations that require knowledge of the previous image contents, such as `copyArea`, `setXOR`, and compositing. These operations can slow rendering and the results might be inconsistent.

7.2.3 Printable Jobs and Pageable Jobs

A `Printable` job provides the simplest way to print. Only one page painter is used; the application provides a single class that implements the `Printable` interface. When it's time to print, the printing system calls the page painter's `print` method to render each page. The pages are requested in order, starting with page index 0. However, the page painter might be asked to render each page several times before it advances to the next page. When the last page has been printed, the page painter's print method returns NO_SUCH_PAGE.

In a `Printable` job:

- All pages use the same page painter and `PageFormat`. If a print dialog is presented, it will not display the number of pages in the document because that information is not available to the printing system.
- The printing system always asks the page painter to print each page in indexed order, starting with the page at index 0. No pages are skipped. For example, if a user asks to print pages 2 and 3 of a document, the page painter will be called with indices 0, 1, and 2. The printing system might request that a page be rendered multiple times before moving to the next page.
- The page painter informs the printing system when the end of the document has been reached.
- All page painters are called in the same thread.
- Some printing systems might not be able to achieve the ideal output. For example, the stack of pages emerging from the printer might be in the wrong order, or the pages might not be collated if multiple copies are requested.

A `Pageable` job is more flexible than a `Printable` job. Unlike the pages in a `Printable` job, pages in a `Pageable` job can differ in layout and implementation.

To manage a `Pageable` job, you can use the `Book` class or implement your own `Pageable` class. Through the `Pageable`, the printing system can determine the number of pages to print, the page painter to use for each page, and the `PageFormat` to use for each page. Applications that need to print documents that have a planned structure and format should use `Pageable` jobs.

In a `Pageable` job:

- Different pages can use different page painters and `PageFormats`.
- The printing system can ask page painters to print pages in an arbitrary order and some pages might be skipped. For example, if a user asks to print pages 2 and 3 of a document, the page painter will be called with indices 1 and 2 and page index 0 will be skipped.
- `Pageable` jobs do not need to know in advance how many pages are in the document. However, unlike `Printable` jobs, they must be able to render pages in any order. There might be gaps in the sequencing and the printing system might request that a page be rendered multiple times before moving to the next page. For example, a request to print pages 2 and 3 of a document might result in a sequence of calls that request pages with indices 2,2,1,1, and 1.

7.2.4 Typical Life-Cycle of a PrinterJob

An application steers the `PrinterJob` object through a sequence of steps to complete a printing job. The simplest sequence used by an application is:

1. Get a new `PrinterJob` object by calling `PrinterJob.getPrinterJob`.
2. Determine what `PageFormat` to use for printing. A default `PageFormat` can be obtained by calling `defaultPage` or you can invoke `pageDialog` to present a dialog box that allows the user to specify a format.
3. Specify the characteristics of the job to be printed to the `PrinterJob`. For a `Printable` job, call `setPrintable`; for a `Pageable` job, call `setPageable`. Note that a `Book` object is ideal for passing to `setPageable`.
4. Specify additional print job properties, such as the number of copies to print or the name of the job to print on the banner page.
5. Call `printDialog` to present a dialog box to the user. This is optional. The contents and appearance of this dialog can vary across different platforms and printers. On most platforms, the user can use this dialog to change the printer selection. If the user cancels the print job, the `printDialog` method returns

FALSE.

6. Call Printerjob.print to print the job. This method in turn calls print on the appropriate page painters.

A job can be interrupted during printing if:

- A PrinterException is thrown—the exception is caught by the print method and the job is halted. A page painter throws a PrinterException if it detects a fatal error.
- PrinterJob.cancel is called—the printing loop is terminated and the job is canceled. The cancel method can be called from a separate thread that displays a dialog box and allows the user to cancel printing by clicking a button in the box.

Pages generated before a print job is stopped might or might not be printed.

The print job is usually not finished when the print method returns. Work is typically still being done by a printer driver, print server, or the printer itself. The state of the PrinterJob object might not reflect the state of the actual job being printed.

Because the state of a PrinterJob changes during its life cycle, it is illegal to invoke certain methods at certain times. For example, calling setPageable after you've called print makes no sense. When illegal calls are detected, the PrinterJob throws a java.lang.IllegalStateException.

7.2.5 Dialogs

The Java Printing API requires that applications invoke user-interface dialogs explicitly. These dialogs might be provided by the platform software (such as Windows) or by a Java™ 2 SDK software implementation. For interactive applications, it is customary to use such dialogs. For production printing applications, however, dialogs are not necessary. For example, you wouldn't want to display a dialog when automatically generating and printing a nightly database report. A print job that requires no user interaction is sometimes called a *silent* print job.

7.2.5.1 Page setup dialog

You can allow the user to alter the page setup information contained in a PageFormat by displaying a page setup dialog. To display the page setup dialog, you call PrinterJob.pageDialog. The page setup dialog is initialized using the parameter passed to `pageDialog`. If the user clicks the OK button in the dialog, the `PageFormat` instance is cloned, altered to reflect the user's selections, and then returned. If the user cancels the dialog, `pageDialog` returns the original unaltered `PageFormat`.

7.2.5.2 Print dialog

Typically, an application presents a print dialog to the user when a print menu item or button is activated. To display this print dialog, you call the `PrinterJob's` `printDialog` method. The user's choices in the dialog are constrained based on the number and format of the pages in the `Printable` or `Pageable` that have been furnished to the `PrinterJob`. If the user clicks OK in the print dialog, `printDialog` returns `TRUE`. If the user cancels the print dialog, `FALSE` is returned and the print job should be considered abandoned.

7.3 Printing with Printables

To provide basic printing support:

1. Implement the `Printable` interface to provide a page painter that can render each page to be printed.
2. Create a `PrinterJob`.
3. Call `setPrintable` to tell the `PrinterJob` how to print your document.
4. Call `print` on the `PrinterJob` object to start the job.

In the following example, a `Printable` job is used to print five pages, each of which displays a green page number. Job control is managed in the `main` method, which obtains and controls the `PrinterJob`. Rendering is performed in the page painter's `print` method.

```
import java.awt.*; import java.awt.print.*;
public class SimplePrint implements Printable
{
  private static Font fnt = new Font("Helvetica",Font.PLAIN,24);

  public static void main(String[] args)
  {
    // Get a PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
```

```
      // Specify the Printable is an instance of SimplePrint
      job.setPrintable(new SimplePrint());
      // Put up the dialog box
      if (job.printDialog())
      {
        // Print the job if the user didn't cancel printing
        try { job.print(); }
        catch (Exception e)
          { /* handle exception */ }
      }
      System.exit(0);
    }

    public int print(Graphics g, PageFormat pf, int pageIndex)
    throws PrinterException
    {
      // pageIndex 0 to 4 corresponds to page numbers 1 to 5.
      if (pageIndex >= 5) return Printable.NO_SUCH_PAGE;
      g.setFont(fnt);
      g.setColor(Color.green);
      g.drawString("Page " + (pageIndex+1), 100, 100);
      return Printable.PAGE_EXISTS;
    }
  }
```

7.3.1 Using Graphics2D for Rendering

You can invoke `Graphics2D` functions in you page painter's print method by first casting the `Graphics` context to a `Graphics2D`.

In the following example, the page numbers are rendered using a red-green gradient. To do this, a `GradientPaint` is set in the `Graphics2D` context.

```
  import java.awt.*; import java.awt.print.*;
  public class SimplePrint2D implements Printable
  {
   private static Font fnt = new Font("Helvetica",Font.PLAIN,24);

    private Paint pnt = new GradientPaint(100f, 100f, Color.red,
                         136f, 100f, Color.green, true);

    public static void main(String[] args)
    {
```

```
      // Get a PrinterJob
      PrinterJob job = PrinterJob.getPrinterJob();
      // Specify the Printable is an instance of SimplePrint2D
      job.setPrintable(new SimplePrint2D());
      // Put up the dialog box
      if (job.printDialog())
      {
        // Print the job if the user didn't cancel printing
        try { job.print(); }
        catch (Exception e) { /* handle exception */ }
      }
    System.exit(0);
    }

    public int print(Graphics g, PageFormat pf, int pageIndex)
    throws PrinterException
    {
      // pageIndex 0 to 4 corresponds to page numbers 1 to 5.
      if (pageIndex >= 5) return Printable.NO_SUCH_PAGE;
      Graphics2D g2 = (Graphics2D) g;
      // Use the font defined above
      g2.setFont(fnt);
      // Use the gradient color defined above
      g2.setPaint(pnt);
      g2.drawString("Page " + (pageIndex+1), 100f, 100f);
      return Printable.PAGE_EXISTS;
    }
  }
```

7.3.2 Printing a File

When a page painter's print method is invoked several times for the same page, it must generate the same output each time.

There are many ways to ensure that repeated requests to render a page yield the same output. For example, to ensure that the same output is generated each time the printing system requests a particular page of a text file, page painter could either store and reuse file pointers for each page or store the actual page data.

In the following example, a "listing" of a text file is printed. The name of the file is passed as an argument to the `main` method. The `PrintListingPainter` class stores the file pointer in effect at the beginning of each new page it is asked to render. When the same page is rendered again, the file pointer is reset to the remembered position.

```
import java.awt.*;
import java.awt.print.*;
import java.io.*;

public class PrintListing
{
  public static void main(String[] args)
  {
    // Get a PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // Ask user for page format (e.g., portrait/landscape)
    PageFormat pf = job.pageDialog(job.defaultPage());
    // Specify the Printable is an instance of
    // PrintListingPainter; also provide given PageFormat
    job.setPrintable(new PrintListingPainter(args[0]), pf);
    // Print 1 copy
    job.setCopies(1);
    // Put up the dialog box
    if (job.printDialog())
    {
      // Print the job if the user didn't cancel printing
      try { job.print(); }
      catch (Exception e) { /* handle exception */ }
    }
    System.exit(0);
  }
}

class PrintListingPainter implements Printable
{
  private RandomAccessFile raf;
  private String fileName;
  private Font fnt = new Font("Helvetica", Font.PLAIN, 10);
  private int rememberedPageIndex = -1;
  private long rememberedFilePointer = -1;
  private boolean rememberedEOF = false;

  public PrintListingPainter(String file)
  {
    fileName = file;
    try
```

```
    {
      // Open file
      raf = new RandomAccessFile(file, "r");
    }
    catch (Exception e) { rememberedEOF = true; }
  }

  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
  try
  {
    // For catching IOException
    if (pageIndex != rememberedPageIndex)
    {
      // First time we've visited this page
      rememberedPageIndex = pageIndex;
      // If encountered EOF on previous page, done
      if (rememberedEOF) return Printable.NO_SUCH_PAGE;
      // Save current position in input file
      rememberedFilePointer = raf.getFilePointer();
    }
    else raf.seek(rememberedFilePointer);
    g.setColor(Color.black);
    g.setFont(fnt);
   int x = (int) pf.getImageableX() + 10;
   int y = (int) pf.getImageableY() + 12;
    // Title line
    g.drawString("File: " + fileName + ", page: " +
                 (pageIndex+1),  x, y);
    // Generate as many lines as will fit in imageable area
    y += 36;
    while (y + 12 < pf.getImageableY()+pf.getImageableHeight())
    {
      String line = raf.readLine();
      if (line == null)
      {
        rememberedEOF = true;
        break;
       }
        g.drawString(line, x, y);
        y += 12;
      }
      return Printable.PAGE_EXISTS;
    }
    catch (Exception e) { return Printable.NO_SUCH_PAGE;}
```

```
        }
    }
```

7.4 Printing with Pageables and Books

`Pageable` jobs are suited for applications that build an explicit representation of a document, page by page. The `Book` class is a convenient way to use `Pageables`, but you can also build your own `Pageable` structures if `Book` does not suit your needs. This section shows you how to use `Book`.

Although slightly more involved, `Pageable` jobs are preferred over `Printable` jobs because the printing system has more flexibility. A major advantage of `Pageables` is that the number of pages in the document is usually known and can be displayed to the user in the print dialog box. This helps the user to confirm that the job is specified correctly or to select a range of pages for printing.

A `Book` represents a collection of pages. The pages in a book do not have to share the same size, orientation, or page painter. For example, a `Book` might contain two letter size pages in portrait orientation and a letter size page in landscape orientation.

When a `Book` is first constructed, it is empty. To add pages to a `Book`, you use the `append` method. This method takes a `PageFormat` object that defines the page's size, printable area, and orientation and a page painter that implements the `Printable` interface.

Multiple pages in a `Book` can share the same page format and painter. The `append` method is overloaded to enable you to add a series of pages that have the same attributes by specifying a third parameter, the number of pages.

If you don't know the total number of pages in a `Book`, you can pass `UNKNOWN_NUMBER_OF_PAGES` to the `append` method. The printing system will then call your page painters in order of increasing page index until one of them returns `NO_SUCH_PAGE`.

The `setPage` method can be used to change a page's page format or painter. The page to be changed is identified by a page index that indicates the page's location in the `Book`.

You call `setPageable` and pass in the `Book` to prepare the print job. The `setPageable` and `setPrintable` methods are mutually exclusive; that is, you should call one or the other but not both when preparing the `PrinterJob`.

7.4.1 Using a Pageable Job

In the following example, a `Book` is used to reproduce the first simple printing example. (Because this case is so simple, there is little benefit in using a `Pageable` job instead of a `Printable` job, but it illustrates the basics of using a `Book`.) Note that you still have to implement the `Printable` interface and perform page rendering in the page painter's `print` method.

```
import java.awt.*;
import java.awt.print.*;

public class SimplePrintBook implements Printable
{
  private static Font fnt = new Font("Helvetica",Font.PLAIN,24);
  public static void main(String[] args)
  {
    // Get a PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // Set up a book
    Book bk = new Book();
    bk.append(new SimplePrintBook(), job.defaultPage(), 5);
    // Pass the book to the PrinterJob
    job.setPageable(bk);
    // Put up the dialog box
    if (job.printDialog())
    {
      // Print the job if the user didn't cancel printing
      try { job.print(); }
      catch (Exception e) { /* handle exception */ }
    }
    System.exit(0);
  }

  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    g.setFont(fnt);
    g.setColor(Color.green);
    g.drawString("Page " + (pageIndex+1), 100, 100);
    return Printable.PAGE_EXISTS;
  }
}
```

7.4.2 Using Multiple Page Painters

In the following example, two different page painters are used: one for a cover page and one for content pages. The cover page is printed in landscape mode and the contents pages are printed in portrait mode.

```
import java.awt.*;
import java.awt.print.*;

public class PrintBook
{
  public static void main(String[] args)
  {
    // Get a PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // Create a landscape page format
    PageFormat pfl = job.defaultPage();
    pfl.setOrientation(PageFormat.LANDSCAPE);
    // Set up a book
    Book bk = new Book();
    bk.append(new PaintCover(), pfl);
    bk.append(new PaintContent(), job.defaultPage(), 2);
    // Pass the book to the PrinterJob
    job.setPageable(bk);
    // Put up the dialog box
    if (job.printDialog())
    {
      // Print the job if the user didn't cancel printing
      try { job.print(); }
      catch (Exception e) { /* handle exception */ }
    }
  System.exit(0);
  }
}

class PaintCover implements Printable
{
  Font fnt = new Font("Helvetica-Bold", Font.PLAIN, 72);

  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    g.setFont(fnt);
    g.setColor(Color.black);
```

```
     int yc = (int) (pf.getImageableY() +
               pf.getImageableHeight()/2);
    g.drawString("Widgets, Inc.", 72, yc+36);
    return Printable.PAGE_EXISTS;
  }
}
class PaintContent implements Printable
{
  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    Graphics2D g2 = (Graphics2D) g;
    int useRed = 0;
   int xo = (int) pf.getImageableX();
   int yo = (int) pf.getImageableY();
   // Fill page with circles or squares, alternating red & green
   for (int x = 0; x+28 < pf.getImageableWidth(); x += 36)
    for (int y = 0; y+28 < pf.getImageableHeight(); y += 36)
    {
      if (useRed == 0) g.setColor(Color.red);
      else g.setColor(Color.green);
      useRed = 1 - useRed;
     if (pageIndex % 2 == 0) g.drawRect(xo+x+4, yo+y+4, 28, 28);
      else g.drawOval(xo+x+4, yo+y+4, 28, 28);
    }
    return   Printable.PAGE_EXISTS;
  }
}
```