
Package java.awt

Interface Index

- Composite
- CompositeContext
- Paint
- PaintContext
- Shape
- Stroke
- Transparency

Class Index

- AlphaComposite
- BasicStroke
- Color
- Font
- GradientPaint
- Graphics2D
- GraphicsConfiguration
- GraphicsDevice
- GraphicsEnvironment
- Rectangle
- TexturePaint

Package java.awt.color

Class Index

Exception Index

Package java.awt.font

Interface Index

Class Index

Package java.awt.geom

Interface Index

- PathIterator

Class Index

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

Exception Index

- IncompatibleTransformException
- NoninvertibleTransformException

Package java.awt.image

Interface Index

- BufferedImageOp
- TileImageConsumer
- TileOp

Class Index

- AffineTransformOp
- BandCombineOp
- BilinearAffineTransformOp
- BufferedImage
- BufferedImageFilter
- ByteBandedTile
- ByteComponentTile
- ByteLookupTable
- BytePackedTile
- ColorConvertOp
- ColorModel
- ComponentColorModel
- ConvolveOp
- DirectColorModel
- IndexColorModel
- IntegerComponentTile
- Kernel
- LookupOp
- LookupTable
- NearestNeighborAffineTransformOp
- PackedColorModel
- RescaleOp
- ShortBandedTile
- ShortComponentTile
- ShortLookupTable
- ThresholdOp
- Tile

Exception Index

Class Hierarchy

- class java.lang.Object
  - class java.awt.geom.AffineTransform (implements java.lang.Cloneable)
  - class java.awt.image.AffineTransformOp (implements java.awt.image.BufferedImageOp , java.awt.image.TileOp)
    - class java.awt.image.BilinearAffineTransformOp
    - class java.awt.image.NearestNeighborAffineTransformOp
  - class java.awt.AlphaComposite (implements java.awt.Composite)
  - class sun.awt.Albert.Area (implements java.awt.Shape)
    - class java.awt.geom.Area
  - class java.awt.image.BandCombineOp (implements java.awt.image.BufferedImageOp, java.awt.image.TileOp)
  - class java.awt.BasicStroke (implements java.awt.Stroke)
  - interface java.awt.image.BufferedImageOp
  - class java.awt.Color (implements java.awt.Paint, java.io.Serializable)
  - class java.awt.image.ColorConvertOp (implements java.awt.image.BufferedImageOp, java.awt.image.TileOp)
  - class java.awt.image.ColorModel (implements java.awt.Transparency)
    - class java.awt.image.ComponentColorModel
    - class java.awt.image.IndexColorModel
    - class java.awt.image.PackedColorModel
      - class java.awt.image.DirectColorModel
  - class java.awt.color.ColorSpace
    - class java.awt.color.ICC_ColorSpace
  - interface java.awt.Composite
  - interface java.awt.CompositeContext
  - class java.awt.image.ConvolveOp (implements java.awt.image.BufferedImageOp, java.awt.image.TileOp)
  - class java.awt.geom.CubicCurve2D (implements java.awt.Shape)
    - class java.awt.geom.CubicCurve2D.Float
  - class java.awt.geom.Dimension2D
  - class java.awt.geom.FlatteningPathIterator (implements java.awt.geom.PathIterator)
  - class java.awt.Font (implements java.io.Serializable)
  - class java.awt.geom.GeneralPath (implements java.awt.Shape)
  - class java.awt.geom.GeneralPathIterator (implements java.awt.geom.PathIterator)
  - class java.awt.font.GlyphMetrics
  - class java.awt.font.GlyphSet
  - class java.awt.GradientPaint (implements java.awt.Paint)
  - class java.awt.Graphics
    - class java.awt.Graphics2D
  - class java.awt.GraphicsConfiguration
  - class java.awt.GraphicsDevice

- class java.awt.GraphicsEnvironment
- class java.awt.color.ICC_Profile
  - class java.awt.color.ICC_ProfileGray
  - class java.awt.color.ICC_ProfileRGB
- class java.awt.Image
  - class java.awt.image.BufferedImage
- class java.awt.image.ImageFilter (implements java.awt.image.ImageConsumer, java.lang.Cloneable)
  - class java.awt.image.BufferedImageFilter (implements java.awt.image.TileImageConsumer, java.lang.Cloneable)
- class java.awt.image.Kernel
- class java.awt.geom.Line2D (implements java.awt.Shape)
  - class java.awt.geom.Line2D.Float
- class java.awt.image.LookupOp (implements java.awt.image.BufferedImageOp, java.awt.image.TileOp)
- class java.awt.image.LookupTable
  - class java.awt.image.ByteLookupTable
  - class java.awt.image.ShortLookupTable
- interface java.awt.font.MultipleMaster
- interface java.awt.font.OpenType
- interface java.awt.Paint (extends java.awt.Transparency)
- interface java.awt.PaintContext
- interface java.awt.geom.PathIterator
- class java.awt.geom.Point2D
  - class java.awt.geom.Point2D.Double
  - class java.awt.geom.Point2D.Float
- class java.awt.geom.QuadCurve2D (implements java.awt.Shape)
  - class java.awt.geom.QuadCurve2D.Float
- class java.awt.geom.RectangularShape (implements java.awt.Shape)
  - class java.awt.geom.Arc2D
    - class java.awt.geom.Arc2D.Float
  - class java.awt.geom.Ellipse2D
    - class java.awt.geom.Ellipse2D.Float
  - class java.awt.geom.Rectangle2D
    - class java.awt.Rectangle (implements java.awt.Shape, java.io.Serializable)
    - class java.awt.geom.Rectangle2D.Double
    - class java.awt.geom.Rectangle2D.Float
  - class java.awt.geom.RoundRectangle2D
    - class java.awt.geom.RoundRectangle2D.Float
- class java.awt.image.RescaleOp (implements java.awt.image.BufferedImageOp, java.awt.image.TileOp)
- interface java.awt.Shape
- interface java.awt.Stroke
- class java.awt.font.StyledString
- class java.awt.font.TextAttributeSet (implements java.text.MutableAttributeSet)
- class java.awt.font.TextHitInfo
- class java.awt.font.TextLayout (implements java.lang.Cloneable)
- class java.awt.TexturePaint (implements java.awt.Paint)
- class java.awt.image.ThresholdOp (implements java.awt.image.BufferedImageOp,

java.awt.image.TileOp)
- class java.lang.Throwable (implements java.io.Serializable)
  - class java.lang.Exception
    - class java.awt.geom.NoninvertibleTransformException
    - class java.lang.RuntimeException
      - class java.awt.geom.IncompatibleTransformException
      - class java.awt.color.NoSuchProfileDataException
      - class java.awt.image.TileFormatException
- class java.awt.image.Tile
  - class java.awt.image.ByteBandedTile
  - class java.awt.image.ByteComponentTile
  - class java.awt.image.BytePackedTile
  - class java.awt.image.IntegerComponentTile
  - class java.awt.image.ShortBandedTile
  - class java.awt.image.ShortComponentTile
- interface java.awt.image.TileImageConsumer (extends java.awt.image.ImageConsumer)
- interface java.awt.image.TileOp
- interface java.awt.Transparency

Interface java.awt.Composite

public interface **Composite**

This interface, together with CompositeContext, defines the methods to compose a draw primitive with the underlying graphics area. The Composite is set in the Graphics2D and thereafter, whenever a shape, text, or an image is drawn, the Composite will combine the source with the colors that have already been drawn, according to pre−defined rules. The classes implementing this interface will provide the rules and a method to create the context for a particular operation. CompositeContext is an environment used by the compositing operation that a Graphics2D object must create prior to the start of the operation. CompositeContext contains various private information and resources needed for a compositing operation. When the CompositeContext is no longer needed it will be disposed by the Graphics2D object to reclaim resources allocated for the operation.

Instances of classes implementing Composite must be immutable (i.e. read−only) because the Graphics2D does not clone these objects when they are set as an attribute with the setComposite method or when the Graphics2D object is itself cloned. This is to avoid undefined behavior of Graphics2D rendering which would result if the Composite object were modified after being set in the Graphics2D state.

See Also:
AlphaComposite, CompositeContext, setComposite

Method Index

o **createContext**(ColorModel, ColorModel)
Create a context for the compositing operation.

Methods

o **createContext**

```
public abstract CompositeContext createContext(ColorModel srcColorModel,
                                               ColorModel dstColorModel)
```

Create a context for the compositing operation. The context contains state that is used to perform the compositing operation. In a multi−threaded environment several contexts may exist simultaneously for a single Composite object.

Parameters:

srcColorModel – The ColorModel of the source.

dstColorModel – The ColorModel of the destination.

Returns:

The CompositeContext object to perform the compositing operation.

Interface java.awt.CompositeContext

public interface **CompositeContext**

This interface defines the encapsulated and optimized environment for a compositing operation. CompositeContext objects maintain state for compositing operations. In a multi−threaded environment, several contexts may exist simultaneously for a single Composite object.

See Also:
Composite

Method Index

o **compose**(Tile, Tile, Tile)
This method composes the two source tiles and places the result in the destination tile.
o **dispose**()
Release resources allocated for context.

Methods

o **dispose**

```
public abstract void dispose()
```

Release resources allocated for context.

o **compose**

```
public abstract void compose(Tile src1,
                             Tile src2,
                             Tile dst)
```

This method composes the two source tiles and places the result in the destination tile. Note that the destination can be the same object as either the first or second source.

Parameters:
src1 – The first source tile for the compositing operation.
src2 – The second source tile for the compositing operation.
dst – The tile where the result of the operation is stored.

Interface java.awt.Paint

public interface **Paint**
extends Transparency

This interface defines how color patterns can be generated for Graphics2D operations. Paint is set in the Graphics2D and thereafter the stroke and fill methods will use the color pattern generated by the implementing Paint class.

Instances of classes implementing Paint must be immutable (i.e. read–only) because the Graphics2D does not clone these objects when they are set as an attribute with the setPaint method or when the Graphics2D object is itself cloned. This is to avoid undefined behavior of Graphics2D rendering which would result if the Paint object were modified after being set in the Graphics2D state.

See Also:
PaintContext, Color, GradientPaint, TexturePaint, setPaint

Method Index

o **createContext**(ColorModel, Rectangle, Rectangle2D, AffineTransform)
Create and return a context used to generate the color pattern.

Methods

o **createContext**

```
 public abstract PaintContext createContext(ColorModel cm,
                                            Rectangle deviceBounds,
                                            Rectangle2D userBounds,
                                            AffineTransform xform)
```

Create and return a context used to generate the color pattern.

Parameters:
cm – ColorModel in which the caller wishes to receive the paint data. This is used only as a hint.
deviceBounds – The Rectangle describing the bounding box in device space of the graphics primitive being rendered.
userBounds – The Rectangle2D describing the bounding box in user space of the graphics primitive being rendered.

xform – The Transform from user space into device space.

Returns:

The PaintContext for generating color patterns.

See Also:

PaintContext

Interface java.awt.PaintContext

public interface **PaintContext**

This interface defines the encapsulated and optimized environment for a paint operation, i.e. generating color patterns in device space for a fill or stroke operation on a Graphics2D. The PaintContext provides the necessary colors for Graphics2D operations in the form of a Tile associated with a ColorModel. The PaintContext maintains state for a particular paint operation. In a multi–threaded environment, several contexts may exist simultaneously for a single Paint object.

See Also:
Paint

Method Index

o **dispose**()
Release the resources allocated for the operation.
o **getColorModel**()
Return the ColorModel of the output.
o **getTile**(int, int, int, int)
Return a Tile containing the colors generated for the graphics operation.

Methods

o **dispose**

```
public abstract void dispose()
```

Release the resources allocated for the operation.

o **getColorModel**

```
public abstract ColorModel getColorModel()
```

Return the ColorModel of the output.

o **getTile**

```
public abstract Tile getTile(int x,
```

```
int y,
int w,
int h)
```

Return a Tile containing the colors generated for the graphics operation.

Parameters:

x,y,w,h – The area in device space for which colors are generated.

Interface java.awt.Shape

public interface **Shape**

The interface for objects which represent some form of geometric shape.

This interface will be revised in the upcoming Java2D project. It is meant to provide a common interface for various existing geometric AWT classes and methods which operate on them. Since it may be superseded or expanded in the future, developers should avoid implementing this interface in their own classes until it is completed in a later release.

Method Index

o **contains**(double, double)
  Test if a given coordinate is inside the boundary of the shape.
o **contains**(double, double, double, double)
  Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
o **contains**(Point2D)
  Test if a given Point is inside the boundary of the shape.
o **contains**(Rectangle2D)
  Test if the interior of the Shape entirely contains the given Rectangle.
o **getBounds**()
  Return the bounding box of the shape.
o **getBounds2D**()
  Return the high precision bounding box of the shape.
o **getPathIterator**(AffineTransform)
  Return an iterator object that iterates along the boundary of the shape and provides access to the geometry of the outline of the shape.
o **getPathIterator**(AffineTransform, double)
  Return an iterator object that iterates along the boundary of the shape and provides access to a flattened view of the geometry of the outline of the shape.
o **intersects**(double, double, double, double)
  Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.
o **intersects**(Rectangle2D)
  Test if the interior of the Shape intersects the interior of a given Rectangle.

Methods

o **getBounds**

```
public abstract Rectangle getBounds()
```

Return the bounding box of the shape.

o **getBounds2D**

```
public abstract Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

o **contains**

```
public abstract boolean contains(double x,
                                 double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **contains**

```
public abstract boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public abstract boolean intersects(double x,
                                   double y,
                                   double w,
                                   double h)
```

Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.

o **intersects**

```
public abstract boolean intersects(Rectangle2D r)
```

Test if the interior of the Shape intersects the interior of a given Rectangle.

o **contains**

```
public abstract boolean contains(double x,
                                 double y,
                                 double w,
                                 double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **contains**

```
public abstract boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

o **getPathIterator**

```
public abstract PathIterator getPathIterator(AffineTransform at)
```

Return an iterator object that iterates along the boundary of the shape and provides access to the geometry of the outline of the shape. An optional affine transform can be specified in which case the coordinates returned in the iteration will be transformed accordingly.

Parameters:

at – an optional AffineTransform to be applied to the coordinates as they are returned in the iteration, or null if the untransformed coordinates are desired.

o **getPathIterator**

```
public abstract PathIterator getPathIterator(AffineTransform at,
                                             double flatness)
```

Return an iterator object that iterates along the boundary of the shape and provides access to a flattened view of the geometry of the outline of the shape. Only SEG_MOVETO, SEG_LINETO, and SEG_CLOSE point types will be returned by the iterator. The amount of subdivision of the curved segments is controlled by the `flatness` parameter which specifies ?REMIND?. An optional affine transform can be specified in which case the coordinates returned in the iteration will be transformed accordingly.

Parameters:

at – an optional AffineTransform to be applied to the coordinates as they are returned in the iteration, or null if the untransformed coordinates are desired.
flatness – the maximum amount that the control points for a given curve can vary from colinear before a subdivided curve is replaced by a straight line connecting the endpoints.

Interface java.awt.Stroke

public interface **Stroke**

This interface allows a Graphics2D object to get a Path representation of the boundaries of a stroking primitive from the path to be stroked. The effect of stroking a path can be compared to drawing a logical pen of an appropriate size and shape along the trajectory of the path. The area where the pen would place ink is the area enclosed by the stroked path.

The stroking primitives of the Graphics2D interface include the drawPath method and any other methods that are implemented in terms of that method such as drawLine, drawRect, drawRoundRect, drawOval, drawArc, drawPolyline, and drawPolygon.

The objects of the classes implementing Stroke must be immutable (i.e. read–only) because the Graphics2D will not clone these objects either when they are set as an attribute with the setStroke method or when the Graphics2D object is itself cloned. If a Stroke object were modified after it was set in the state of the Graphics2D then the behavior of subsequent rendering would be undefined.

See Also:
    BasicStroke, setStroke

Method Index

o **createStrokedShape**(Shape)
    Returns a path which encloses the area that should be painted when the path is stroked according to the rules defined by the object implementing the Stroke interface.

Methods

o **createStrokedShape**

```
public abstract Shape createStrokedShape(Shape p)
```

Returns a path which encloses the area that should be painted when the path is stroked according to the rules defined by the object implementing the Stroke interface.

Parameters:
    p – The path that should be stroked.
Returns:

The stroked path.

Interface java.awt.Transparency

public interface **Transparency**

This interface defines the common transparency modes for implementing classes.

Variable Index

o **BITMASK**
The transparency object represents image data that is guaranteed to be either completely opaque (alpha value of 1.0) or completely transparent (alpha value of 0.0).
o **OPAQUE**
The transparency object represents image data that is guaranteed to be completely opaque (all pixels have an alpha value of 1.0).
o **TRANSLUCENT**
The transparency object represents image data that contains or may contain arbitrary alpha values.

Method Index

o **getTransparency**()
Returns the transparency type.

Variables

o **OPAQUE**

```
public static final int OPAQUE
```

The transparency object represents image data that is guaranteed to be completely opaque (all pixels have an alpha value of 1.0).

o **BITMASK**

```
public static final int BITMASK
```

The transparency object represents image data that is guaranteed to be either completely opaque (alpha value of 1.0) or completely transparent (alpha value of 0.0).

o **TRANSLUCENT**

```
public static final int TRANSLUCENT
```

The transparency object represents image data that contains or may contain arbitrary alpha values.

Methods

o **getTransparency**

```
public abstract int getTransparency()
```

Returns the transparency type.

Class java.awt.AlphaComposite

```
java.lang.Object
   |
   +----java.awt.AlphaComposite
```

---

public final class **AlphaComposite**
extends Object
implements Composite

This class implements the basic alpha compositing rules for combining source and destination pixels to achieve blending and transparency effects with graphics and images. The rules implemented by this class are a subset of the Porter–Duff rules described in T. Porter and T. Duff, "Compositing Digital Images", SIGGRAPH 84, 253–259.

If any input does not have an alpha channel, an alpha value of 1.0 (completely opaque) is assumed for all pixels.

A constant alpha value can also be specified to be multiplied with the alpha value of the source pixels.

The following abbreviations are used in the description of the rules:

- Cs = one of the Color components of the Source Pixel.
- Cd = one of the Color components of the Destination Pixel.
- As = Alpha component of the Source Pixel.
- Ad = Alpha component of the Destination Pixel.
- Fs = fraction of the Source Pixel which contributes to the output.
- Fd = fraction of the input Destination Pixel which contributes to the output.

The color and alpha components produced by the compositing operation are calculated as follows:

```
        Cd = Cs*Fs + Cd*Fd
        Ad = As*Fs + Ad*Fd
```

where Fs and Fd are specified by each rule. The above equations assume that both source and destination pixels have the color components premultiplied by the alpha component. Similarly, the equations expressed in the definitions of compositing rules below assume premultiplied alpha.

For performance reasons, it is preferable that Tiles passed to the compose method of a CompositeContext object created by the AlphaComposite class have premultiplied data. However, if either source or destination Tiles are not premultiplied, appropriate conversions will be performed before and after the compositing operation.

The resulting alpha of the compositing operation is stored in the destination if the destination has an alpha channel. Otherwise, the resulting color is divided by the resulting alpha before being stored in the destination and the alpha is discarded (if the alpha value is 0.0, the color values are set to 0.0).

See Also:
Composite, CompositeContext

Variable Index

o **CLEAR**
Porter−Duff Clear rule.
o **DST_IN**
Porter−Duff Destination In Source rule.
o **DST_OUT**
Porter−Duff Destination Held Out By Source rule.
o **DST_OVER**
Porter−Duff Destination Over Source rule.
o **SRC**
Porter−Duff Source rule.
o **SRC_IN**
Porter−Duff Source In Destination rule.
o **SRC_OUT**
Porter−Duff Source Held Out By Destination rule.
o **SRC_OVER**
Porter−Duff Source Over Destination rule.

Method Index

o **createContext**(ColorModel, ColorModel)
Creates a context for the compositing operation.
o **equals**(Object)
o **getAlpha**()
Returns the additional alpha value that was given when this AlphaComposiste instance was created.
o **getInstance**(int)
Creates an AlphaComposite object with the given rule.
o **getInstance**(int, float)
Creates an AlphaComposite object with the given rule and the constant alpha to multiply with the alpha of the source.
o **getRule**()
Returns the compositing rule that was specified when this AlphaComposiste instance was created.

Variables

o **CLEAR**

```
public static final int CLEAR
```

Porter−Duff Clear rule. Both the color and the alpha of destination are cleared. Neither the source nor the destination is used as input.

Fs = 0 and Fd = 0, thus:

```
Cd = 0
Ad = 0
```

o **SRC**

```
public static final int SRC
```

Porter−Duff Source rule. The source is copied to the destination. The destination is not used as input.

Fs = 1 and Fd = 0, thus:

```
Cd = Cs
Ad = As
```

o **SRC_OVER**

```
public static final int SRC_OVER
```

Porter−Duff Source Over Destination rule. The source is composited over the destination.

Fs = 1 and Fd = (1−As), thus:

```
Cd = Cs + Cd*(1-As)
Ad = As + Ad*(1-As)
```

o **DST_OVER**

```
public static final int DST_OVER
```

Porter−Duff Destination Over Source rule. The destination is composited over the source and the result replaces the destination.

Fs = (1−Ad) and Fd = 1, thus:

```
Cd = Cs*(1-Ad) + Cd
Ad = As*(1-Ad) + Ad
```

o **SRC_IN**

```
public static final int SRC_IN
```

Porter−Duff Source In Destination rule. The part of the source lying inside of the destination replaces the destination.

Fs = Ad and Fd = 0, thus:

```
Cd = Cs*Ad
Ad = As*Ad
```

o **DST_IN**

```
public static final int DST_IN
```

Porter−Duff Destination In Source rule. The part of the destination lying inside of the source replaces the destination.

Fs = 0 and Fd = As, thus:

```
Cd = Cd*As
Ad = Ad*As
```

o **SRC_OUT**

```
public static final int SRC_OUT
```

Porter−Duff Source Held Out By Destination rule. The part of the source lying outside of the destination replaces the destination.

Fs = (1−Ad) and Fd = 0, thus:

```
Cd = Cs*(1-Ad)
Ad = As*(1-Ad)
```

o **DST_OUT**

```
public static final int DST_OUT
```

Porter−Duff Destination Held Out By Source rule. The part of the destination lying outside of the source replaces the destination.

Fs = 0 and Fd = (1−As), thus:

```
Cd = Cd*(1-As)
Ad = Ad*(1-As)
```

Methods

o **getInstance**

```
public static AlphaComposite getInstance(int rule)
```

Creates an AlphaComposite object with the given rule.

Parameters:
rule – Rule for the composition.

o **getInstance**

```
public static AlphaComposite getInstance(int rule,
                                         float alpha)
```

Creates an AlphaComposite object with the given rule and the constant alpha to multiply with the alpha of the source. The source is multiplied with the given alpha before being composited with the destination.

Parameters:
rule – Rule for the composition.
alpha – The constant alpha to be multiplied with the alpha of the source. Alpha must be a floating point number in the range 0.0 – 1.0.

o **createContext**

```
public CompositeContext createContext(ColorModel srcColorModel,
                                      ColorModel dstColorModel)
```

Creates a context for the compositing operation. The context contains state that is used in performing the compositing operation.

Parameters:
srcColorModel – The ColorModel of the source.
dstColorModel – The ColorModel of the destination.
Returns:
The CompositeContext object to be used to perform compositing operations.

o **getAlpha**

```
public float getAlpha()
```

Returns the additional alpha value that was given when this AlphaComposiste instance was created. If this instance was created without the addtional alpha value, the value 1.0 is returned.

o **getRule**

```
public int getRule()
```

Returns the compositing rule that was specified when this AlphaComposiste instance was created.

o **equals**

```
public boolean equals(Object obj)
```

Overrides:
equals in class Object

Class java.awt.BasicStroke

```
java.lang.Object
   |
   +----java.awt.BasicStroke
```

---

public class **BasicStroke**
extends Object
implements Stroke

Defines a basic set of rendering attributes for stroked graphics primitives. These attributes describe the shape of a pen drawn along the trajectory of the path and the decorations applied where path segments are joined and where they begin and end. These attributes include:

width
: The width of the pen measured perpendicular to its trajectory.

end caps
: The decoration of the path where line segments begin and end.

line joins
: The decoration of the path between connected line segments.

dash attributes
: The definition of how to make a dash pattern along the trajectory of the path by creating new segments which alternate between opaque and transparent sections.

Variable Index

o **CAP_BUTT**
End unclosed subpaths and dash segments with no added decoration.

o **CAP_ROUND**
End unclosed subpaths and dash segments with a round decoration with radius equal to half of the line width.

o **CAP_SQUARE**
End unclosed subpaths and dash segments with a square projection that extends beyond the end of the segment to a distance equal to half of the line width.

o **JOIN_BEVEL**
Join line segments by connecting the outer corners of their wide outlines with a straight segment.

o **JOIN_MITER**
Join line segments by extending their outside edges until they meet.
o **JOIN_ROUND**
Join line segments by rounding off the corner at a radius of half the line width.

Constructor Index

o **BasicStroke**()
Construct a new stroke with defaults for all attributes.
o **BasicStroke**(float, int, int)
Construct a new stroke with the specified attributes, but no dashing.
o **BasicStroke**(float, int, int, float)
Construct a new stroke with the specified attributes, but no dashing.
o **BasicStroke**(float, int, int, float, float[], float)
Construct a new stroke with the specified attributes.

Method Index

o **createStrokedShape**(Shape)
Returns the stroked path of a given path.
o **equals**(Object)
o **getDashArray**()
Return the array representing the lengths of the dash segments.
o **getDashPhase**()
Returns the dash phase.
o **getEndCap**()
Returns the end cap style.
o **getLineJoin**()
Returns the line join style.
o **getLineWidth**()
Returns the line width.
o **getMiterLimit**()
Returns the limit of miter joins.

Variables

o **JOIN_MITER**

```
public static final int JOIN_MITER
```

Join line segments by extending their outside edges until they meet.

o **JOIN_ROUND**

```
public static final int JOIN_ROUND
```

Join line segments by rounding off the corner at a radius of half the line width.

o **JOIN_BEVEL**

```
public static final int JOIN_BEVEL
```

Join line segments by connecting the outer corners of their wide outlines with a straight segment.

o **CAP_BUTT**

```
public static final int CAP_BUTT
```

End unclosed subpaths and dash segments with no added decoration.

o **CAP_ROUND**

```
public static final int CAP_ROUND
```

End unclosed subpaths and dash segments with a round decoration with radius equal to half of the line width.

o **CAP_SQUARE**

```
public static final int CAP_SQUARE
```

End unclosed subpaths and dash segments with a square projection that extends beyond the end of the segment to a distance equal to half of the line width.

Constructors

o **BasicStroke**

```
public BasicStroke(float width,
                   int cap,
                   int join,
                   float miterlimit,
                   float dash[],
                   float dash_phase)
```

Construct a new stroke with the specified attributes.

Parameters:
width – The width of the stroke.
cap – The style of the ends of a stroke.
join – The style to join strokes together.
miterlimit – The limit to trim the miter join.
dash – The array representing the dashing pattern.
dash_phase – The offset to start the dashing pattern.

o **BasicStroke**

```
public BasicStroke(float width,
                   int cap,
                   int join,
                   float miterlimit)
```

Construct a new stroke with the specified attributes, but no dashing.

Parameters:
width – The width of the stroke.
cap – The style of the ends of a stroke.
join – The style to join strokes together.
miterlimit – The limit to trim the miter join.

o **BasicStroke**

```
public BasicStroke(float width,
                   int cap,
                   int join)
```

Construct a new stroke with the specified attributes, but no dashing. The miter limit parameter is unnecessary in cases where the default is allowable or the line joins are not specified as JOIN_MITER.

Parameters:
width – The width of the stroke.
cap – The style of the ends of a stroke.
join – The style to join strokes together.

o **BasicStroke**

```
public BasicStroke()
```

Construct a new stroke with defaults for all attributes. The default attributes are a line width of 1.0, CAP_SQUARE, JOIN_MITER, a miter limit of 10.0, and no dashing.

Methods

o **createStrokedShape**

```
public Shape createStrokedShape(Shape p)
```

Returns the stroked path of a given path.

Parameters:
p – The path that should be stroked.
Returns:
The stroked path.
See Also:

createStrokedShape

o **getLineWidth**

```
public float getLineWidth()
```

Returns the line width. Line width is represented in user space.

Returns:
The line width of the stroke.

o **getEndCap**

```
public int getEndCap()
```

Returns the end cap style.

Returns:
The end cap style of the stroke.

o **getLineJoin**

```
public int getLineJoin()
```

Returns the line join style.

Returns:
The line join style of the stroke.

o **getMiterLimit**

```
public float getMiterLimit()
```

Returns the limit of miter joins.

Returns:
The limit of miter joins.

o **getDashArray**

```
public float[] getDashArray()
```

Return the array representing the lengths of the dash segments. Alternate entries in the array represent the user space lengths of the opaque and transparent segments of the dashes. As the pen moves along the path to be stroked, the user space distance that the pen travels will be accumulated to index into the dash array. The pen will be opaque when its current cumulative distance maps to an odd element of the dash array and transparent otherwise.

Returns:

The dash array.

o **getDashPhase**

```
public float getDashPhase()
```

Returns the dash phase. The dash phase is a user space distance that is used to initialize the path distance accumulation variable before the first segment of the first subpath is stroked.

Returns:

the dash phase.

o **equals**

```
public boolean equals(Object obj)
```

Overrides:

equals in class Object

Class java.awt.Color

```
java.lang.Object
   |
   +----java.awt.Color
```

---

public class **Color**
extends Object
implements Paint, Serializable

A class to encapsulate colors in the default sRGB color space or colors in arbitrary color spaces identified by a ColorSpace. Every color has an implicit alpha value of 1.0 or an explicit one provided in the constructor. When constructing a Color with an explicit alpha or getting the color/alpha components of a Color, the color components are never premultiplied by the alpha component.

The default color space for the Java 2D API is sRGB, a proposed standard RGB color space. For further information on sRGB, see http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html .

Eventually this class should supersede java.awt.Color, but for now it is defined to be part of the java.java2d package, so we can generate documentation for a single package for review.

See Also:
ColorSpace

Variable Index

o **black**
The color black.
o **blue**
The color blue.
o **cyan**
The color cyan.
o **darkGray**
The color dark gray.
o **gray**
The color gray.
o **green**

The color green.

o **lightGray**
  The color light gray.

o **magenta**
  The color magenta.

o **orange**
  The color orange.

o **pink**
  The color pink.

o **red**
  The color red.

o **white**
  The color white.

o **yellow**
  The color yellow.

Constructor Index

o **Color**(ColorSpace, float[], float)
  Creates a color in the color space of the supplied ColorSpace with the color components specified in the float array and the specified alpha.

o **Color**(float, float, float)
  Creates an opaque sRGB color with the specified red, green, and blue values in the range (0.0 – 1.0).

o **Color**(float, float, float, float)
  Creates an sRGB color with the specified red, green, blue, and alpha values in the range (0.0 – 1.0).

o **Color**(int)
  Creates an opaque sRGB color with the specified combined RGB value consisting of the red component in bits 16−23, the green component in bits 8−15, and the blue component in bits 0−7.

o **Color**(int, boolean)
  Creates an sRGB color with the specified combined RGBA value consisting of the alpha component in bits 24−31, the red component in bits 16−23, the green component in bits 8−15, and the blue component in bits 0−7.

o **Color**(int, int, int)
  Creates an opaque sRGB color with the specified red, green, and blue values in the range (0 – 255).

o **Color**(int, int, int, int)
  Creates an sRGB color with the specified red, green, blue, and alpha values in the range (0 – 255).

Method Index

o **brighter**()
  Creates a brighter version of this color.

o **createContext**(ColorModel, Rectangle, Rectangle2D, AffineTransform)

Create and return a PaintContext used to generate a solid color pattern.
o **darker**()
Creates a darker version of this color.
o **decode**(String)
Converts a string to an integer and returns the specified color.
o **equals**(Object)
Determines whether another object is equal to this color.
o **getAlpha**()
Returns the alpha component.
o **getBlue**()
Returns the blue component.
o **getColor**(String)
Finds a color in the system properties.
o **getColor**(String, Color)
Finds a color in the system properties.
o **getColor**(String, int)
Finds a color in the system properties.
o **getColorComponents**(ColorSpace, float[])
Returns a float array containing the color components (no alpha) of the Color, in the ColorSpace specified by the cspace parameter.
o **getColorComponents**(float[])
Returns a float array containing the color components (no alpha) of the Color, in the ColorSpace of the color.
o **getColorSpace**()
Returns the ColorSpace of the Color.
o **getComponents**(ColorSpace, float[])
Returns a float array containing the color and alpha components of the Color, in the ColorSpace specified by the cspace parameter.
o **getComponents**(float[])
Returns a float array containing the color and alpha components of the Color, in the ColorSpace of the Color.
o **getGreen**()
Returns the green component.
o **getHSBColor**(float, float, float)
A static Color factory for generating a Color object from HSB values.
o **getRed**()
Returns the red component.
o **getRGB**()
Returns the RGB value representing the color in the default sRGB ColorModel.
o **getRGBA**()
Returns the RGBA value representing the color in the default sRGB ColorModel.
o **getRGBColorComponents**(float[])
Returns a float array containing the color components (no alpha) of the Color, in the default sRGB color space.
o **getRGBComponents**(float[])
Returns a float array containing the color and alpha components of the Color, as represented in the default sRGB color space.
o **getTransparency**()
Return the transparency mode for this Color.

o **hashCode**()
Computes the hash code for this color.
o **HSBtoRGB**(float, float, float)
Returns the RGB value defined by the default sRGB ColorModel, of the color corresponding to the given HSB color components.
o **RGBtoHSB**(int, int, int, float[])
Returns the HSB values corresponding to the color defined by the red, green, and blue components, assumed to be in the default sRGB color space.
o **toString**()
Returns the String representation of this Color.

Variables

o **white**

```
public static final Color white
```

The color white. In the default sRGB space.

o **lightGray**

```
public static final Color lightGray
```

The color light gray. In the default sRGB space.

o **gray**

```
public static final Color gray
```

The color gray. In the default sRGB space.

o **darkGray**

```
public static final Color darkGray
```

The color dark gray. In the default sRGB space.

o **black**

```
public static final Color black
```

The color black. In the default sRGB space.

o **red**

```
public static final Color red
```

The color red. In the default sRGB space.

o **pink**

```
public static final Color pink
```

The color pink. In the default sRGB space.

o **orange**

```
public static final Color orange
```

The color orange. In the default sRGB space.

o **yellow**

```
public static final Color yellow
```

The color yellow. In the default sRGB space.

o **green**

```
public static final Color green
```

The color green. In the default sRGB space.

o **magenta**

```
public static final Color magenta
```

The color magenta. In the default sRGB space.

o **cyan**

```
public static final Color cyan
```

The color cyan. In the default sRGB space.

o **blue**

```
public static final Color blue
```

The color blue. In the default sRGB space.

Constructors

o **Color**

```
public Color(int r,
             int g,
```

```
              int b)
```

Creates an opaque sRGB color with the specified red, green, and blue values in the range (0 – 255). The actual color used in rendering will depend on finding the best match given the color space available for a given output device. Alpha is defaulted to 255.

Parameters:
r – the red component
g – the green component
b – the blue component
See Also:
getRed, getGreen, getBlue, getRGB

o **Color**

```
public Color(int r,
             int g,
             int b,
             int a)
```

Creates an sRGB color with the specified red, green, blue, and alpha values in the range (0 – 255).

Parameters:
r – the red component
g – the green component
b – the blue component
a – the alpha component
See Also:
getRed, getGreen, getBlue, getAlpha, getRGBA

o **Color**

```
public Color(int rgb)
```

Creates an opaque sRGB color with the specified combined RGB value consisting of the red component in bits 16–23, the green component in bits 8–15, and the blue component in bits 0–7. The actual color used in rendering will depend on finding the best match given the color space available for a given output device. Alpha is defaulted to 255.

Parameters:
rgb – the combined RGB components
See Also:
getRGBdefault, getRed, getGreen, getBlue, getRGB

o **Color**

```
public Color(int rgba,
             boolean hasalpha)
```

Creates an sRGB color with the specified combined RGBA value consisting of the alpha component in bits 24−31, the red component in bits 16−23, the green component in bits 8−15, and the blue component in bits 0−7. If the hasalpha argument is False, alpha is defaulted to 255.

Parameters:
rgba − the combined RGBA components
hasalpha − true if the alpha bits are valid, false otherwise
See Also:
getRGBdefault, getRed, getGreen, getBlue, getAlpha, getRGBA

o **Color**

```
public Color(float r,
             float g,
             float b)
```

Creates an opaque sRGB color with the specified red, green, and blue values in the range (0.0 − 1.0). Alpha is defaulted to 1.0. The actual color used in rendering will depend on finding the best match given the color space available for a given output device.

Parameters:
r − the red component
g − the green component
b − the blue component
See Also:
getRed, getGreen, getBlue, getRGB

o **Color**

```
public Color(float r,
             float g,
             float b,
             float a)
```

Creates an sRGB color with the specified red, green, blue, and alpha values in the range (0.0 − 1.0). The actual color used in rendering will depend on finding the best match given the color space available for a given output device.

Parameters:
r − the red component
g − the green component
b − the blue component
a − the alpha component
See Also:
getRed, getGreen, getBlue, getAlpha, getRGBA

o **Color**

```
public Color(ColorSpace cspace,
             float components[],
```

```
float alpha)
```

Creates a color in the color space of the supplied ColorSpace with the color components specified in the float array and the specified alpha. The number of components is determined by the color space type of the ColorSpace (e.g. RGB requires 3 components, CMYK requires 4, etc.).

Parameters:
cspace − The ColorSpace to be used to interpret the components
components − An arbitrary number of color components that is compatible with the ColorSpace
alpha − Alpha value

See Also:
getComponents, getColorComponents

Methods

o **getRed**

```
public int getRed()
```

Returns the red component. In the range 0−255 in the default sRGB space.

See Also:
getRGB

o **getGreen**

```
public int getGreen()
```

Returns the green component. In the range 0−255 in the default sRGB space.

See Also:
getRGB

o **getBlue**

```
public int getBlue()
```

Returns the blue component. In the range 0−255 in the default sRGB space.

See Also:
getRGB

o **getAlpha**

```
public int getAlpha()
```

Returns the alpha component. In the range 0−255.

See Also:
getRGBA

o **getRGB**

```
public int getRGB()
```

Returns the RGB value representing the color in the default sRGB ColorModel. (Bits 24−31 are 0xff, 16−23 are red, 8−15 are green, 0−7 are blue).

See Also:
getRGBdefault, getRed, getGreen, getBlue

o **getRGBA**

```
public int getRGBA()
```

Returns the RGBA value representing the color in the default sRGB ColorModel. (Bits 24−31 are alpha, 16−23 are red, 8−15 are green, 0−7 are blue).

See Also:
getRGBdefault, getRed, getGreen, getBlue, getAlpha

o **brighter**

```
public Color brighter()
```

Creates a brighter version of this color.

This method applies an arbitrary scale factor to each of the three RGB components of the color to create a brighter version of the same color. Although `brighter` and `darker` are inverse operations, the results of a series of invocations of these two methods may be inconsistent because of rounding errors.

Returns:
a new `Color` object, a brighter version of this color.

See Also:
darker

o **darker**

```
public Color darker()
```

Creates a darker version of this color.

This method applies an arbitrary scale factor to each of the three RGB components of the color to create a darker version of the same color. Although `brighter` and `darker` are inverse operations, the results of a series of invocations of these two methods may be inconsistent because of rounding errors.

Returns:
a new `Color` object, a darker version of this color.
See Also:
brighter

o **hashCode**

```
public int hashCode()
```

Computes the hash code for this color.

Returns:
a hash code value for this object.
Overrides:
hashCode in class Object

o **equals**

```
public boolean equals(Object obj)
```

Determines whether another object is equal to this color.

The result is `true` if and only if the argument is not `null` and is a `Color` object that has the same red, green, and blue values as this object.

Parameters:
obj – the object to compare with.
Returns:
`true` if the objects are the same; `false` otherwise.
Overrides:
equals in class Object

o **toString**

```
public String toString()
```

Returns the String representation of this Color.

Overrides:
toString in class Object

o **decode**

```
public static Color decode(String nm) throws NumberFormatException
```

Converts a string to an integer and returns the specified color. This method handles string formats that are used to represent octal and hexidecimal numbers.

Parameters:
nm − a string that represents a color as a 24−bit integer.
Returns:
the new color
Throws: NumberFormatException
if the specified string cannot be interpreted as a decimal, octal, or hexidecimal integer.
See Also:
decode

o **getColor**

```
public static Color getColor(String nm)
```

Finds a color in the system properties.

The argument is treated as the name of a system property to be obtained. The string value of this property is then interpreted as an integer which is then converted to a color.

If the specified property is not found, or could not be parsed as an integer, then `null` is returned.

Parameters:
nm − the name of the color property
Returns:
the color value of the property.
See Also:
getProperty, getInteger, Color

o **getColor**

```
public static Color getColor(String nm,
                             Color v)
```

Finds a color in the system properties.

The first argument is treated as the name of a system property to be obtained. The string value of this property is then interpreted as an integer which is then converted to a color.

If the specified property is not found, or cannot be parsed as an integer, then the color specified by the second argument is returned instead.

Parameters:
nm − the name of the color property

v – the default color value.

Returns:

the color value of the property.

See Also:

getProperty, getInteger, Color

o **getColor**

```
public static Color getColor(String nm,
                             int v)
```

Finds a color in the system properties.

The first argument is treated as the name of a system property to be obtained. The string value of this property is then interpreted as an integer which is then converted to a color.

If the specified property is not found, or could not be parsed as an integer, then the integer value `v` is used instead, and is converted to a color.

Parameters:

nm – the name of the color property.

v – the default color value, as an integer.

Returns:

the color value of the property.

See Also:

getProperty, getInteger, Color

o **HSBtoRGB**

```
public static int HSBtoRGB(float hue,
                           float saturation,
                           float brightness)
```

Returns the RGB value defined by the default sRGB ColorModel, of the color corresponding to the given HSB color components.

Parameters:

hue – the hue component of the color

saturation – the saturation of the color

brightness – the brightness of the color

See Also:

getRGBdefault, getRGB

o **RGBtoHSB**

```
public static float[] RGBtoHSB(int r,
                               int g,
                               int b,
                               float hsbvals[])
```

Returns the HSB values corresponding to the color defined by the red, green, and blue components, assumed to be in the default sRGB color space.

Parameters:
r – the red component of the color
g – the green component of the color
b – the blue component of the color
hsbvals – the array to be used to return the 3 HSB values, or null

Returns:
the array used to store the results [hue, saturation, brightness]

See Also:
getRGBdefault, getRGB

o **getHSBColor**

```
public static Color getHSBColor(float h,
                                float s,
                                float b)
```

A static Color factory for generating a Color object from HSB values.

Parameters:
h – the hue component
s – the saturation of the color
b – the brightness of the color

Returns:
the Color object for the corresponding RGB color in the default sRGB space.

o **getRGBComponents**

```
public float[] getRGBComponents(float componentarray[])
```

Returns a float array containing the color and alpha components of the Color, as represented in the default sRGB color space. If componentarray is null, an array of length 4 is created for the return value. Otherwise, componentarray must have length 4 or greater, and it will be filled in with the components and returned.

Returns:
the RGBA components in a float array

o **getRGBColorComponents**

```
public float[] getRGBColorComponents(float componentarray[])
```

Returns a float array containing the color components (no alpha) of the Color, in the default sRGB color space. If componentarray is null, an array of length 3 is created for the return value. Otherwise, componentarray must have length 3 or greater, and it will be filled in with the components and returned.

Returns:
the RGB components in a float array

o **getComponents**

```
public float[] getComponents(float componentarray[])
```

Returns a float array containing the color and alpha components of the Color, in the ColorSpace of the Color. If componentarray is null, an array with length equal to the number of components in the associated ColorSpace plus one is created for the return value. Otherwise, componentarray must have at least this length, and it will be filled in with the components and returned.

Returns:
the color and alpha components in a float array

o **getColorComponents**

```
public float[] getColorComponents(float componentarray[])
```

Returns a float array containing the color components (no alpha) of the Color, in the ColorSpace of the color. If componentarray is null, an array with length equal to the number of components in the associated ColorSpace is created for the return value. Otherwise, componentarray must have at least this length, and it will be filled in with the components and returned.

Returns:
the color components in a float array

o **getComponents**

```
public float[] getComponents(ColorSpace cspace,
                             float componentarray[])
```

Returns a float array containing the color and alpha components of the Color, in the ColorSpace specified by the cspace parameter. If componentarray is null, an array with length equal to the number of components in cspace plus one is created for the return value. Otherwise, componentarray must have at least this length, and it will be filled in with the components and returned.

Returns:
the color and alpha components in a float array

o **getColorComponents**

```
public float[] getColorComponents(ColorSpace cspace,
                                  float componentarray[])
```

Returns a float array containing the color components (no alpha) of the Color, in the

ColorSpace specified by the cspace parameter. If componentarray is null, an array with length equal to the number of components in cspace is created for the return value. Otherwise, componentarray must have at least this length, and it will be filled in with the components and returned.

Returns:
the color components in a float array

o **getColorSpace**

```
public ColorSpace getColorSpace()
```

Returns the ColorSpace of the Color.

o **createContext**

```
public synchronized PaintContext createContext(ColorModel cm,
                                               Rectangle r,
                                               Rectangle2D r2d,
                                               AffineTransform xform)
```

Create and return a PaintContext used to generate a solid color pattern. This enables a Color object to be used as an argument to any method requiring an object implementing the Paint interface.

See Also:
Paint, PaintContext, setPaint

o **getTransparency**

```
public int getTransparency()
```

Return the transparency mode for this Color. This is required in to implement the Paint interface.

See Also:
Paint, Transparency, createContext

Class java.awt.Font

```
java.lang.Object
   |
   +----java.awt.Font
```

---

public class **Font**
extends Object
implements Serializable

This class represents fonts. The capabilities of this class have been extended over the java.awt.Font class in JDK 1.1 and earlier releases to provide developers the ability to utilize more sophisticated typographic features.

It is important to present the concepts behind using the words character and glyph separately. A **character** is a symbol that represents items like letters and numbers in a given writing system, for example *lowercase−g*. When a given character is drawn, a shape now represents this character. This shape is called a **glyph**.

Chararcter encoding is a conversion table that maps character codes to glyph codes in the font. The character encoding used in the Java 2D API is Unicode. For more information on Unicode you may visit the site http://www.unicode.org.

Characters and glyphs do not have one−to−one correspondence. As an example *lowercase−a acute* can be represented by two glyphs: *lowercase−a* and *acute*. Another example is ligatures such as *ligature −fi* which is a single glyph representing two characters: *f* and *i*.

A font is a collection of glyphs. A font may have many faces, e.g. heavy, medium, oblique, gothic and regular. All of these faces have similar typographic design.

There are three different names that you can get from a Font object. The *logical font name*is the same as used by java.awt.Font in JDK 1.1 and earlier releases. The java.awt.Toolkit.getFontList() method returns a short list of these logical names which are mapped onto specific fonts available on specific platforms. The *font face name*, or just *font name* for short, is the name of a particular font face, like Helvetica Bold. The *family name* is the name of the font family that determines the typographic design across several faces, like Helvetica. The font face name is the one that should be used to specify fonts. This name signifies actual fonts in the host system, and does not identify font names with shape of font characters as the logical font name does.

The Font class represents an instance of a font face from a collection of font faces that are present in the system resources of the host system. As examples, Helvetica Bold and Courier Bold Italic are font faces. There can be several Font objects associated with a font face, each differing in size, style, transform and font features. GraphicsEnvironment.getAllFonts() returns an array of all font faces available in the system. These font faces are returned as Font objects with a size of 1, identity transform and default font features. These base fonts can then be used to derive new Font objects with varying sizes, styles, transforms and font features via the deriveFont methods in this class.

See Also:
getFontList, getFonts, getAllFonts

Variable Index

o **BOLD**
The bold style constant.
o **CENTER_BASELINE**
o **DEFAULT**
o **HANGING_BASELINE**
o **ITALIC**
The italicized style constant.
o **name**
The logical name of this font.
o **PLAIN**
The plain style constant.
o **ROMAN_BASELINE**
o **size**
The point size of this font, rounded to integer.
o **style**
The style of the font, as passed to the constructor.

Constructor Index

o **Font**(AttributeSet)
Create a new font with the specified attributes
o **Font**(FontObject)
o **Font**(String, int, int)
Creates a new font from the specified name, style and point size.

Method Index

o **canDisplay**(char)
Checks if this font has a glyph for the specified character.
o **canDisplayUpTo**(CharacterIterator)
Indicates whether a string is displayable by this Font.
o **canDisplayUpTo**(String)

Indicates whether a string is displayable by this Font.

o **decode**(String)
: Returns the specified font using the name passed in.

o **deriveFont**(AffineTransform)
: Creates a new Font object by replicating the current Font object with a new transform associated with it.

o **deriveFont**(float)
: Creates a new Font object by replicating the current Font object with a new size associated with it.

o **deriveFont**(int)
: Creates a new Font object by replicating the current Font object with a new style associated with it.

o **deriveFont**(int, AffineTransform)
: Creates a new Font object by replicating the current Font object with a new style, transform and font attributes associated with it.

o **deriveFont**(int, float)
: Creates a new Font object by replicating the current Font object with a new style, size and font attributes associated with it.

o **deriveFont**(TextAttributeSet)
: Creates a new Font object by replicating the current Font object with a new set of font attributes associated with it.

o **equals**(Object)
: Compares this object to the specified object.

o **finalize**()

o **getActualAttributes**()
: Returns the requested attributes that this font is actually using.

o **getAscent**()
: Returns the ascent of the font above the roman baseline.

o **getAttributes**()
: Returns an array of font attributes available in this font.

o **getAvailableAttributes**()
: Returns the names of all the attributes supported by this font.

o **getBaselineFor**(char)

o **getBaselineOffset**(byte)
: Returns the offset from the roman baseline of the font to the specified baseline.

o **getBaselineOffsetsFor**(char)
: Return a list of relative offsets for the different baselines.

o **getBestFontFor**(AttributedCharacterIterator)

o **getCapHeight**()
: Returns the cap height of this font.

o **getDescent**()
: Returns the descent of the font above the roman baseline.

o **getFamily**()
: Returns the family name of the font (for example, Helvetica).

o **getFamily**(Locale)
: Returns the family name of the font (for example, Helvetica), localized for the given Locale.

o **getFont**(String)
: Returns a font from the system properties list.

o **getFont**(String, Font)

Returns the specified font from the system properties list.

o **getFontName**()

Returns the font face name of the font (for example, Helvetica Bold).

o **getFontName**(Locale)

Returns the font face name of the font (for example, Helvetica Fett) localized for the specified locale.

o **getGlyphJustificationInfo**(int)

Return justification info for the glyph specified by glyphCode.

o **getGlyphMetrics**(int)

Returns the metrics information for a glyph specified by a glyph code.

o **getGlyphOutline**(int, float, float)

Returns the outline description of a glyph specified by a glyph code.

o **getGlyphSet**(AttributedCharacterIterator, int, int, byte, int[], byte[])

o **getGlyphSet**(CharacterIterator, byte, int[], byte[])

Converts characters in the charArray to font glyph codes.

o **getGlyphSet**(String)

Converts all characters in the String Object to font glyph codes.

o **getItalicAngle**()

Returns the italic angle of this font.

o **getLeading**()

Returns the leading for this font;

o **getMaxAdvance**()

Returns the maximum advance of any glyph in this font.

o **getMaxBounds2D**()

Returns the maximum bounding box of this font.

o **getMaxLSB**()

Returns the maximum left side bearing of any glyph in this font.

o **getMaxRSB**()

Returns the maximum right side bearing of any glyph in this font.

o **getMetrics**(GlyphSet, AffineTransform)

o **getMissingGlyphCode**()

o **getName**()

Returns the logical name of the font.

o **getNumGlyphs**()

Returns number of glyphs in the font.

o **getOutline**(GlyphSet, AffineTransform, float, float)

o **getPeer**()

Gets the peer of the font.

o **getPSName**()

Returns the postscript name of the font.

o **getRequestedAttributes**()

Returns the attribute set used to create this font.

o **getSize**()

Returns the point size of the font, rounded to integer.

o **getSize2D**()

Returns the point size of the font in float.

o **getStyle**()

Returns the style of the font.

o **getTransform**()

Returns the transform associated with this font.

o **getXHeight**()

Returns the x−height for this font.

o **handleApplyJustification**(GlyphSet, float[], int, boolean[])

Return a new glyphset after applying justification.

o **hashCode**()

Returns a hashcode for this font.

o **isBold**()

Indicates whether the font's style is bold.

o **isItalic**()

Indicates whether the font's style is italic.

o **isPlain**()

Returns true if the font is plain.

o **isVerticalLine**()

Return true if the font lays out glyphs along a vertical baseline.

o **toString**()

Converts this object to a String representation.

Variables

o **DEFAULT**

```
public static final Font DEFAULT
```

o **PLAIN**

```
public static final int PLAIN
```

The plain style constant.

o **BOLD**

```
public static final int BOLD
```

The bold style constant. This can be combined with the other style constants (except PLAIN) for mixed styles.

o **ITALIC**

```
public static final int ITALIC
```

The italicized style constant. This can be combined with the other style constants (except PLAIN) for mixed styles.

o **ROMAN_BASELINE**

```
public static final byte ROMAN_BASELINE
```

o **CENTER_BASELINE**

```
public static final byte CENTER_BASELINE
```

o **HANGING_BASELINE**

```
public static final byte HANGING_BASELINE
```

o **name**

```
protected String name
```

The logical name of this font.

o **style**

```
protected int style
```

The style of the font, as passed to the constructor. This may be PLAIN, BOLD, ITALIC, or BOLD+ITALIC.

o **size**

```
protected int size
```

The point size of this font, rounded to integer.

Constructors

o **Font**

```
public Font(String name,
            int style,
            int size)
```

Creates a new font from the specified name, style and point size.

Parameters:
name – the font name. This can be a logical font name or a font face name.
style – the style constant for the font.
size – the point size of the font.
See Also:
getFontList, getFonts, getAllFonts

o **Font**

```
public Font(FontObject fontObject)
```

o **Font**

```
public Font(AttributeSet attributes)
```

Create a new font with the specified attributes

Methods

o **getPeer**

```
public FontPeer getPeer()
```

Gets the peer of the font.

Returns:
the peer of the font.

o **getTransform**

```
public AffineTransform getTransform()
```

Returns the transform associated with this font.

o **getFamily**

```
public String getFamily()
```

Returns the family name of the font (for example, Helvetica). Use getName to get the logical name of the font. Use getFontName to get the font face name of the font.

See Also:
getName, getFontName

o **getFamily**

```
public String getFamily(Locale l)
```

Returns the family name of the font (for example, Helvetica), localized for the given Locale. Use getFontName to get the font face name of the font.

Parameters:
l – Locale for which to get the family name.
See Also:
getFontName, Locale

o **getPSName**

```
public String getPSName()
```

Returns the postscript name of the font. Use getFamily to get the family name of the font. Use getFontName to get the font face name of the font.

o **getName**

```
public String getName()
```

Returns the logical name of the font. Use getFamily to get the family name of the font. Use getFontName to get the font face name of the font.

See Also:
getFamily, getFontName

o **getFontName**

```
public String getFontName()
```

Returns the font face name of the font (for example, Helvetica Bold). Use getFamily to get the family name of the font. Use getName to get the logical name of the font.

See Also:
getFamily, getName

o **getFontName**

```
public String getFontName(Locale l)
```

Returns the font face name of the font (for example, Helvetica Fett) localized for the specified locale. Use getFamily to get the family name of the font.

Parameters:
l – Get the localized font face name for this locale.
See Also:
getFamily, Locale

o **getStyle**

```
public int getStyle()
```

Returns the style of the font. This may be PLAIN, BOLD, ITALIC, or BOLD+ITALIC.

See Also:
isPlain, isBold, isItalic

o **getSize**

```
public int getSize()
```

Returns the point size of the font, rounded to integer. Most users are familiar with the idea of using *point size* to specify the size of glyphs in a font. This point size defines a measurement between the baseline of one line to the baseline of the following line in a single spaced text document. The point size is based on *typographic points*, approximately 1/72 of an inch.

The Java2D API adopts the convention that one point is equivalent to one unit in user coordinates. When using a normalized transform for converting user space coordinates to device space coordinates (see GraphicsConfiguration.getDefaultTransform() and GraphicsConfiguration.getNormalizingTransform()), 72 user space units equal 1 inch in device space. In this case one point is 1/72 of an inch.

See Also:
getSize2D, getDefaultTransform, getNormalizingTransform

o **getSize2D**

```
public float getSize2D()
```

Returns the point size of the font in float.

See Also:
getSize

o **isPlain**

```
public boolean isPlain()
```

Returns true if the font is plain.

See Also:
getStyle

o **isBold**

```
public boolean isBold()
```

Indicates whether the font's style is bold.

Returns:
`true` if the font is bold; `false` otherwise.
See Also:
getStyle

o **isItalic**

```
public boolean isItalic()
```

Indicates whether the font's style is italic.

Returns:

`true` if the font is italic; `false` otherwise.

See Also:

getStyle

o **getFont**

```
public static Font getFont(String nm)
```

Returns a font from the system properties list.

Parameters:

nm – the property name.

o **decode**

```
public static Font decode(String str)
```

Returns the specified font using the name passed in.

Parameters:

str – the name.

o **getFont**

```
public static Font getFont(String nm,
                           Font font)
```

Returns the specified font from the system properties list.

Parameters:

nm – the property name.
font – a default font to return if property 'nm' is not defined.

o **hashCode**

```
public int hashCode()
```

Returns a hashcode for this font.

Returns:

a hashcode value for this font.

Overrides:

hashCode in class Object

o **equals**

```
public boolean equals(Object obj)
```

Compares this object to the specified object.

Parameters:
obj – the object to compare with.
Returns:
true if the objects are the same; false otherwise.
Overrides:
equals in class Object

o **toString**

```
public String toString()
```

Converts this object to a String representation.

Returns:
a string representation of this object
Overrides:
toString in class Object

o **getNumGlyphs**

```
public int getNumGlyphs()
```

Returns number of glyphs in the font. Glyph codes for the font will range from 0 to getNumGlyphs() – 1.

o **getMissingGlyphCode**

```
public int getMissingGlyphCode()
```

o **getGlyphMetrics**

```
public GlyphMetrics getGlyphMetrics(int glyphCode)
```

Returns the metrics information for a glyph specified by a glyph code.

Parameters:
glyphCode – the glyph to get metrics for.
Returns:
a GlyphMetrics object with metric information for the glyph.

o **getBaselineFor**

```
public byte getBaselineFor(char c)
```

o **getBaselineOffsetsFor**

```
public float[] getBaselineOffsetsFor(char c)
```

Return a list of relative offsets for the different baselines.

Large fonts can support different writing systems, and each system can have its own set of preferred baseline offsets. The character argument determines the writing system to use.

These all are relative to the most common baseline used with this font (the one from which ascent and descent are measured). Negative values are up on horizontal lines, and right on vertical lines.

Parameters:
c – a character used to identify the writing system

See Also:
getBaselineFor, ROMAN_BASELINE, CENTER_BASELINE, HANGING_BASELINE

o **getGlyphOutline**

```
public Shape getGlyphOutline(int glyphCode,
                             float x,
                             float y)
```

Returns the outline description of a glyph specified by a glyph code.

Parameters:
glyphCode – the glyph for which to get the outline.

Returns:
a Shape object representing the outline of the glyph.

o **getMetrics**

```
public float[] getMetrics(GlyphSet glyphs,
                          AffineTransform tx)
```

o **getOutline**

```
public Shape getOutline(GlyphSet glyphs,
                        AffineTransform tx,
                        float x,
                        float y)
```

o **getAttributes**

```
public AttributeSet getAttributes()
```

Returns an array of font attributes available in this font. Attributes include things like ligatures and glyph substitution.

Returns:
a FontAttribute array.

o **getAvailableAttributes**

```
public String[] getAvailableAttributes()
```

Returns the names of all the attributes supported by this font. These attributes may be used to derive other fonts.

o **getGlyphJustificationInfo**

```
public GlyphJustificationInfo getGlyphJustificationInfo(int glyphCode)
```

Return justification info for the glyph specified by glyphCode. If left is true, return info for the left side of the glyph, otherwise for the right. If grow is true, return info for growing a line, otherwise for shrinking it. !!! unify into one set of information per glyph? Then it's the accessing code's problem...

o **getRequestedAttributes**

```
public AttributeSet getRequestedAttributes()
```

Returns the attribute set used to create this font. NOTE: This method may be moved into another class.

o **getActualAttributes**

```
public TextAttributeSet getActualAttributes()
```

Returns the requested attributes that this font is actually using. NOTE: This method may be moved into another class;

o **deriveFont**

```
public Font deriveFont(int style,
                       float size)
```

Creates a new Font object by replicating the current Font object with a new style, size and font attributes associated with it.

Parameters:
style – the style for the new Font.
size – the size in float for the new Font.
attributes – an array of AttributeSets enabled for the new Font.

Returns:
a new Font object.

o **deriveFont**

```
public Font deriveFont(int style,
                       AffineTransform trans)
```

Creates a new Font object by replicating the current Font object with a new style, transform and font attributes associated with it.

Parameters:
style – the style for the new Font.
trans – the AffineTransform associated with the new Font.
attributes – an array of AttributeSets enabled for the new Font.
Returns:
a new Font object.

o **deriveFont**

```
public Font deriveFont(float size)
```

Creates a new Font object by replicating the current Font object with a new size associated with it.

Parameters:
size – the size in float for the new Font.
Returns:
a new Font object.

o **deriveFont**

```
public Font deriveFont(AffineTransform trans)
```

Creates a new Font object by replicating the current Font object with a new transform associated with it.

Parameters:
trans – the AffineTransform associated with the new Font.
Returns:
a new Font object.

o **deriveFont**

```
public Font deriveFont(int style)
```

Creates a new Font object by replicating the current Font object with a new style associated with it.

Parameters:
style – the style for the new Font.
Returns:
a new Font object.

o **deriveFont**

```
public Font deriveFont(TextAttributeSet attributes)
```

Creates a new Font object by replicating the current Font object with a new set of font attributes associated with it.

Parameters:
attributes – an array of AttributeSets enabled for the new Font.

Returns:
a new Font object.

o **isVerticalLine**

```
public boolean isVerticalLine()
```

Return true if the font lays out glyphs along a vertical baseline. The default is false.

o **handleApplyJustification**

```
public GlyphSet handleApplyJustification(GlyphSet gs,
                                         float deltas[],
                                         int index,
                                         boolean flags[])
```

Return a new glyphset after applying justification.

This is called by GlyphSet.

Subclassers can override this to build a new glyphset in any way appropriate to the requested assignment of space. The original glyphset may not be modified. In particular changing the values in the arrays of the glyphset will cause incorrect results. A new glyphset should be created if any modification occurs.

Parameters:
gs – the original glyphset
deltas – an array of amounts by which to change the left and right sides of each glyph, in visual order.
index – the starting position in the deltas array, it corresponds to the left side of the leftmost glyph.
flags – on entry, indicates whether the returned glyphset is allowed to cause rejustification, e.g. if a ligature can be formed or broken. On exit, indicates whether the returned glyphset will require rejustification.

Returns:
a new glyphset.

o **getGlyphSet**

```
public GlyphSet getGlyphSet(AttributedCharacterIterator context,
```

```
                              int start,
                              int limit,
                              byte baseline,
                              int order[],
                              byte levels[])
```

o **getGlyphSet**

```
public GlyphSet getGlyphSet(CharacterIterator textLine,
                              byte baseline,
                              int visualOrder[],
                              byte levels[])
```

Converts characters in the charArray to font glyph codes. Glyph substitution may be performed.

Parameters:
text – an CharacterIterator.
Returns:
a GlyphSet containing a collection of glyphs and glyph positions.
See Also:
CharacterIterator

o **getGlyphSet**

```
public GlyphSet getGlyphSet(String str)
```

Converts all characters in the String Object to font glyph codes. Glyph substitution may be performed.

Parameters:
str – a String object.
Returns:
a GlyphSet containing a collection of glyphs and glyph positions.

o **canDisplay**

```
public boolean canDisplay(char c)
```

Checks if this font has a glyph for the specified character.

Parameters:
c – a unicode character code.
Returns:
true if the font can display the character.

o **canDisplayUpTo**

```
public int canDisplayUpTo(String str)
```

Indicates whether a string is displayable by this Font. For strings with Unicode encoding, it is important to know if a given Font can display the string. This method returns an offset into the String str which is the first character the Font cannot display without using the missing glyph code. If the Font can display all characters, −1 is returned.

Parameters:
str – a String object.
Returns:
an offset into the String object that can be displayed by this font.

o **canDisplayUpTo**

```
public int canDisplayUpTo(CharacterIterator iter)
```

Indicates whether a string is displayable by this Font. For strings with Unicode encoding, it is important to know if a given Font can display the string. This method returns an offset into the String str which is the first character the Font cannot display without using the missing glyph code . If the Font can display all characters, −1 is returned.

Parameters:
text – a CharacterIterator object.
Returns:
an offset into the String object that can be displayed by this font.

o **getBaselineOffset**

```
public float getBaselineOffset(byte baselineType)
```

Returns the offset from the roman baseline of the font to the specified baseline.

o **getAscent**

```
public float getAscent()
```

Returns the ascent of the font above the roman baseline.

o **getDescent**

```
public float getDescent()
```

Returns the descent of the font above the roman baseline.

o **getLeading**

```
public float getLeading()
```

Returns the leading for this font;

o **getMaxAdvance**

```
public float getMaxAdvance()
```

Returns the maximum advance of any glyph in this font.

o **getMaxLSB**

```
public float getMaxLSB()
```

Returns the maximum left side bearing of any glyph in this font.

o **getMaxRSB**

```
public float getMaxRSB()
```

Returns the maximum right side bearing of any glyph in this font.

o **getXHeight**

```
public float getXHeight()
```

Returns the x−height for this font.

o **getCapHeight**

```
public float getCapHeight()
```

Returns the cap height of this font.

o **getMaxBounds2D**

```
public Rectangle2D getMaxBounds2D()
```

Returns the maximum bounding box of this font.

o **getItalicAngle**

```
public float getItalicAngle()
```

Returns the italic angle of this font.

o **getBestFontFor**

```
public static Font getBestFontFor(AttributedCharacterIterator text)
```

o **finalize**

```
protected void finalize() throws Throwable
```

Overrides:
finalize in class Object

Class java.awt.GradientPaint

```
java.lang.Object
   |
   +----java.awt.GradientPaint
```

public class **GradientPaint**
extends Object
implements Paint

This class provides a way to fill a shape with a linear color gradient pattern. Given two points in user space, P1 with the color C1 and P2 with the color C2, the color on the P1, P2 connecting line is changed from C1 to C2 proportionally to the distance to P1. All the points on the extended line on the P1 side have the color C1 and on the P2 side have the color C2. Any point P not on the extended P1, P2 connecting line has the color of the point P' that is the perpendicular projection of P on the extended P1, P2 connecting line.

See Also:
Paint, setPaint

Constructor Index

o **GradientPaint**(float, float, Color, float, float, Color)
Constructs a GradientPaint object.
o **GradientPaint**(Point2D, Color, Point2D, Color)
Constructs a GradientPaint object.

Method Index

o **createContext**(ColorModel, Rectangle, Rectangle2D, AffineTransform)
Creates and returns a context used to generate the color pattern.
o **getTransparency**()
Return the transparency mode for this GradientPaint.

Constructors

o **GradientPaint**

```
public GradientPaint(float x1,
                     float y1,
                     Color color1,
                     float x2,
                     float y2,
                     Color color2)
```

Constructs a GradientPaint object.

Parameters:
x1,y1 – Coordinates of the 1st point in user space.
color1 – Color at the 1st point.
x2,y2 – Coordinates of the 2nd point in user space.
color2 – Color at the 2nd point.

o **GradientPaint**

```
public GradientPaint(Point2D pt1,
                     Color color1,
                     Point2D pt2,
                     Color color2)
```

Constructs a GradientPaint object.

Parameters:
pt1 – Coordinates of the 1st point in user space.
color1 – Color at the 1st point.
pt2 – Coordinates of the 2nd point in user space.
color2 – Color at the 2nd point.

Methods

o **createContext**

```
public PaintContext createContext(ColorModel cm,
                                  Rectangle deviceBounds,
                                  Rectangle2D userBounds,
                                  AffineTransform xform)
```

Creates and returns a context used to generate the color pattern.

Parameters:
cm – ColorModel in which the caller wishes to receive the paint data. This is used only as a hint.
deviceBounds – The rectangle describing the bounding box in device space of the graphics primitive being rendered.

userBounds – The rectangle describing the bounding box in user space of the graphics primitive being rendered.
xform – The Transform from user space into device space.

Returns:

The PaintContext for generating color patterns.

See Also:

PaintContext

o **getTransparency**

```
public int getTransparency()
```

Return the transparency mode for this GradientPaint.

See Also:

Transparency

Class java.awt.Graphics2D

```
java.lang.Object
   |
   +----java.awt.Graphics
           |
           +----java.awt.Graphics2D
```

public abstract class **Graphics2D**
extends Graphics

This is the fundamental class for 2D rendering in Java. This class extends the original java.awt.Graphics class to provide more sophisticated control over geometry, coordinate transformation, color management, and text layout.

Coordinate Spaces

All coordinates that are given to a Graphics2D object are treated as being in a virtual coordinate system which is called the User Coordinate Space, or User Space. The Graphics2D object contains a Transform object as part of its rendering state that defines how to convert coordinates from this user space to coordinates in the Device Coordinate Space, or Device Space.

Coordinates in device space usually refer to individual device pixels and are aligned in the infinitely thin gaps between these pixels. Some Graphics2D objects may be used to capture rendering operations for storage into a graphics metafile for playback on a concrete device of unknown physical resolution at a later time. Since the resolution may not be known when the rendering operations are captured, the Transform will be set up to transform user coordinates to a virtual device space that typically approximates the expected resolution of the target device, though further transformation may need to be applied at playback time if the estimate is incorrect.

Some of the operations performed by the rendering attribute objects may operate in the device space, but all methods that are defined on the Graphics2D object work with user space coordinates.

The default transform when a Graphics2D object is created will be specified by the GraphicsConfiguration for the target of the Graphics2D (a Component or Image). This default transform will map the user space coordinate system to screen and printer device coordinates such that the origin maps to the upper left hand corner of the target region of the device with increasing X coordinates extending to the right and increasing Y coordinates extending downward. The scaling of the default transform for screen devices will be set to identity for screen devices. The scaling of the default transform for printers and other high resolution devices will be set to approximate 72 user space coordinates per device inch. For image buffers, the default transform will be the Identity transform.

Rendering Outline

Rendering in the Java 2D API can be described by a 4−step conceptual process controlled by the various rendering attributes in the Graphics2D object. The renderer may optimize many of these steps, either by caching the results for future calls, by collapsing multiple virtual steps into a single operation, or by recognizing various attributes as common simple cases that can be eliminated by modifying other parts of the operation.

As a behavioral reference, the steps can be described in the following way:

1. Determine where to render. This step can be further broken down according to the type of rendering operation as follows:
   Shape operations
   1. If the operation is a draw(Shape) operation, then the shape is converted into a shape to fill using the createStrokedShape() method on the current Stroke.
   2. The shape is transformed from user space to device space using the current Transform.
   3. The outline of the Shape is extracted using a PathIterator using the Shape.getPathIterator() method.
   4. If the Graphics2D object is not able to handle the curved segments that a PathIterator object can return, then it may call the alternat getPathIterator() method which flattens the shape.

   Text operations
   1. The set of glyphs required to render the indicated string are determined as follows:
      - If the argument is a String, then the current Font is asked to convert the Unicode characters in the String into a GlyphSet with whatever basic layout algorithm the font implements.
      - If the argument is a StyledString, the StyledString is asked to convert itself to a GlyphSet according to its embedded font attributes. The StyledString may implement more sophisticated glyph layout algorithms that perform Unicode bi−directional layout adjustments automatically for multiple fonts of differing writing directions.
      - If the argument is a GlyphSet, then the GlyphSet object already contains the appropriate font−specific glyph codes with explicit coordinates for the position of each glyph.
   2. The current Font is queried to obtain outlines for the indicated glyphs. These outlines are treated as shapes in user space relative to the position of each glyph that was determined in step 1.
   3. The outline of the characters are filled as indicated above under Shape operations

   Image operations
   1. The bounding box of the source image defines the region of interest in a coordinate system local to the Image object, called Image Space.
   2. If an optional transform is supplied to the drawImage() method call, then this transform is used to transform the bounding box from image space to user space, otherwise the bounding box is treated as if it already was in user space.
   3. The bounding box of the source image is then transformed from user space into device

space using the current transform of the Graphics2D. Note that the result of transforming the bounding box will not necessarily result in a rectangular region in device space and for some custom transforms may not even result in a quadrilateral region.

2. Constrain the rendering operation to the current Clip. The clip was originally specified by a shape in user space and was converted into device space by the Transform in effect at the time it was modified.
3. Determine what colors to render as the renderer scan converts the associated shape in device space.
    - For image operations, the colors are determined from the Image object itself sampled according to the source to destination coordinate mapping specified by the current transform and the optional image transform.
    - For all other operations, the current Paint or Color object is queried for a PaintContext which specifies what colors are to be rendered in device space.
4. Apply the colors to the destination drawing surface using the current Composite object.

Default values

The default values for the various rendering attributes are:

Color
: The color of the Component.

Font
: The font of the Component.

Stroke
: A square pen with a linewidth of 1, no dashing, miter segment joins and non−projecting end caps.

Transform
: The default for the GraphicsConfiguration of the Component.

Composite
: The Porter−Duff Source Over Destination rule.

Clip
: No rendering clip, output is clipped to the Component

Rendering compatibility issue

The rendering model presented by the pre−Java 2D API Graphics class was based on a pixelization model that specified that coordinates were infinitely thin, lying between the pixels, and that drawing operations were performed by dragging a one−pixel square Pen along the path hanging down and to the right of the anchor point on the path.

The behavior of these operations given the model presented here is unclear if the current Stroke specifies a wide pen, or if there is a non−identity Transform being applied.

That rendering model was consistent with the capabilities of most of the existing class of platform renderers that needed to resolve integer coordinates to a discrete pen that must fall completely on a given number of pixels. Since the Java 2D API is capable of driving an antialiasing renderer it no longer makes sense to choose a bias direction for a wide pen since sub−pixel positioning of the pen can be made visible to the user via the blending that occurs along the edges of the traversal of the pen. It is thus, no longer true that a 1−pixel wide pen must choose to fall on pixel N as opposed to pixel N+1; it can now fall partially on both pixels.

When antialiasing is turned off with the ANTIALIAS_OFF hint, the underlying renderer may still need to apply a bias in order to resolve the conflict of which pixel to modify when the pen is exactly straddling a pixel boundary (such as when it is drawn along an integer coordinate in device space). But when antialiasing is enabled, no biasing of the pen needs to occur and so no bias should be explicitly specified by the rendering model.

Thus, there remains the problem of how to define the operation of the legacy rendering operations inherited from the Graphics class to be compatible with existing rendering behavior and to be predictable and unsurprising under all possible settings of the latest rendering attributes.

To provide predictability we will define operations for all of the legacy methods that map onto the more general draw(Shape) and fill(Shape) methods. Such a specification will thus be clear how it extends based on various settings of Stroke and Transform attributes, and rendering hints.

To provide compatibility with previous rendering behavior it is important to specify a definition that performs identically under the default attribute settings. In particular, the default Stroke is a BasicStroke with a width of 1 and no dashing, and the default Transform for screen drawing is an Identity transform. Unfortunately, the default antialiasing rendering hint is ANTIALIAS_DEFAULT which may be enabled on some implementations and not on others. This means that the definition of these operations needs to be invariant under antialiasing.

We now define the following generalizations of the various legacy methods which can be shown to be identical to the previously specified behavior under the default attributes:

- For fill operations, which include fillRect, fillRoundRect, fillOval, fillArc, fillPolygon, and clearRect, we could create a GeneralPath, gp, that traverses the same integer coordinates and call:

  ```
  fill(gp);
  ```

- For draw operations, which include drawLine, drawRect, drawRoundRect, drawOval, drawArc, drawPolyline, and drawPolygon, we could create a GeneralPath, gp, that traverses the same integer coordinates and call:

  ```
  AffineTransform at = new AffineTransform(1f,0f,0f,1f,0.5f,0.5f);
  draw(at.createTransformedShape(bp));
  ```

- The draw3DRect and fill3DRect methods were implemented in terms of the drawLine and fillRect methods and their behavior is thus extended as well.

The existing Graphics class defined only the single setColor method to control the color to be painted. Since the Java 2D API extends the Color object to implement the new Paint interface, the existing setColor method is now a convenience method for setting the current Paint to a Color object.

`setColor(c)` is thus equivalent to `setPaint(c)`.

The existing Graphics class defined two methods for controlling how colors were applied to the destination. The setPaintMode() method will now be implemented as a convenience method to set the default Composite, equivalent to `setComposite(new AlphaComposite(SRC_OVER))`. The setXORMode(Color xorcolor) method will now be implemented as a convenience method to set a special Composite object which ignores the Alpha components of source colors and sets the destination color to the value:

```
dstpixel = (PixelOf(srccolor) ^ PixelOf(xorcolor) ^ dstpixel);
```

Variable Index

o **ANTIALIAS_DEFAULT**
Rendering is done with the platform default antialiasing mode.
o **ANTIALIAS_OFF**
Rendering is done without antialiasing.
o **ANTIALIAS_ON**
Rendering is done with antialiasing.
o **ANTIALIASING**
Antialiasing hint category.
o **RENDER_DEFAULT**
The platform default algorithms are chosen for rendering.
o **RENDER_QUALITY**
Appropriate rendering algorithms are chosen with a preference for output quality.
o **RENDER_SPEED**
Appropriate rendering algorithms are chosen with a preference for output speed.
o **RENDERING**
Rendering hint category.

Constructor Index

o **Graphics2D**()
Constructs a new Graphics2D object.

Method Index

o **clip**(Shape)
Intersects the current clip with the interior of the specified Shape and sets the current clip to the resulting intersection.
o **draw**(Shape)
Strokes the outline of a Shape using the settings of the current graphics state.
o **drawImage**(BufferedImage, BufferedImageOp, int, int)
Draws a BufferedImage that is filtered with a BufferedImageOp.

- o **drawImage**(Image, AffineTransform, ImageObserver)
  Draws an image, applying a transform from image space into user space before drawing.
- o **drawString**(GlyphSet, float, float)
  Draws a GlyphSet.
- o **drawString**(String, float, float)
  Draws a string of text.
- o **drawString**(StyledString, float, float)
  Draws a StyledString.
- o **drawString**(TextLayout, float, float)
- o **fill**(Shape)
  Fills the interior of a Shape using the settings of the current graphics state.
- o **getBackground**()
  Returns the background color used for clearing a region.
- o **getComposite**()
  Returns the current Composite in the Graphics2D state.
- o **getDeviceConfiguration**()
  Returns the device configuration associated with this Graphics2D.
- o **getPaint**()
  Returns the current Paint in the Graphics2D state.
- o **getRenderingHints**(int)
  Returns the preferences for the rendering algorithms.
- o **getStroke**()
  Returns the current Stroke in the Graphics2D state.
- o **getTransform**()
  Returns the current Transform in the Graphics2D state.
- o **hit**(Rectangle, Shape, boolean)
  Checks to see if the outline of a Shape intersects the specified Rectangle in device space.
- o **hitString**(Rectangle, StyledString, float, float)
  Checks to see if the StyledString intersects the specified Rectangle in device space.
- o **setBackground**(Color)
  Sets the background color in this context used for clearing a region.
- o **setComposite**(Composite)
  Sets the Composite in the current graphics state.
- o **setPaint**(Paint)
  Sets the Paint in the current graphics state.
- o **setRenderingHints**(int, int)
  Sets the preferences for the rendering algorithms.
- o **setStroke**(Stroke)
  Sets the Stroke in the current graphics state.
- o **setTransform**(AffineTransform)
  Sets the Transform in the current graphics state.
- o **transform**(AffineTransform)
  Composes a Transform object with the transform in this Graphics2D according to the rule last−specified−first−applied.

Variables

o **ANTIALIASING**

```
public static final int ANTIALIASING
```

Antialiasing hint category.

o **ANTIALIAS_ON**

```
public static final int ANTIALIAS_ON
```

Rendering is done with antialiasing.

o **ANTIALIAS_OFF**

```
public static final int ANTIALIAS_OFF
```

Rendering is done without antialiasing.

o **ANTIALIAS_DEFAULT**

```
public static final int ANTIALIAS_DEFAULT
```

Rendering is done with the platform default antialiasing mode.

o **RENDERING**

```
public static final int RENDERING
```

Rendering hint category.

o **RENDER_SPEED**

```
public static final int RENDER_SPEED
```

Appropriate rendering algorithms are chosen with a preference for output speed.

o **RENDER_QUALITY**

```
public static final int RENDER_QUALITY
```

Appropriate rendering algorithms are chosen with a preference for output quality.

o **RENDER_DEFAULT**

```
public static final int RENDER_DEFAULT
```

The platform default algorithms are chosen for rendering.

Constructors

o **Graphics2D**

```
protected Graphics2D()
```

Constructs a new Graphics2D object. Since Graphics2D is an abstract class, and since it must be customized by subclasses for different output devices, Graphics2D objects cannot be created directly. Instead, Graphics2D objects must be obtained from another Graphics2D object or created by a Component.

See Also:
getGraphics, create

Methods

o **draw**

```
public abstract void draw(Shape s)
```

Strokes the outline of a Shape using the settings of the current graphics state. The rendering attributes applied include the clip, transform, paint or color, composite and stroke attributes.

Parameters:
s – The shape to be drawn.
See Also:
setStroke, setPaint, setColor, transform, setTransform, clip, setClip, setComposite

o **drawImage**

```
public abstract void drawImage(Image img,
                               AffineTransform xform,
                               ImageObserver obs)
```

Draws an image, applying a transform from image space into user space before drawing. The transformation from user space into device space is done with the current transform in the Graphics2D. The given transformation is applied to the image before the transform attribute in the Graphics2D state is applied. The rendering attributes applied include the clip, transform, and composite attributes. Note that the result is undefined, if the given transform is noninvertible.

Parameters:
img – The image to be drawn.
xform – The transformation from image space into user space.
obs – The image observer to be notified as more of the image is converted.
See Also:

transform, setTransform, setComposite, clip, setClip

o **drawImage**

```
public abstract void drawImage(BufferedImage img,
                               BufferedImageOp op,
                               int x,
                               int y)
```

Draws a BufferedImage that is filtered with a BufferedImageOp. The rendering attributes applied include the clip, transform and composite attributes. This is equivalent to:

```
img1 = op.filter(img, null);
drawImage(img1, new AffineTransform(1f,0f,0f,1f,x,y), null);
```

Parameters:
op – The filter to be applied to the image before drawing.
img – The BufferedImage to be drawn.
x,y – The location in user space where the image should be drawn.

See Also:
transform, setTransform, setComposite, clip, setClip

o **drawString**

```
public abstract void drawString(String s,
                                float x,
                                float y)
```

Draws a string of text. The rendering attributes applied include the clip, transform, paint or color, font and composite attributes.

Parameters:
s – The string to be drawn.
x,y – The coordinates where the string should be drawn.

See Also:
setPaint, setColor, setFont, transform, setTransform, setComposite, clip, setClip

o **drawString**

```
public abstract void drawString(StyledString s,
                                float x,
                                float y)
```

Draws a StyledString. The rendering attributes applied include the clip, transform, paint or color, and composite attributes. A Font is associated with each character in the StyledString.

Parameters:
s – The StyledString to be drawn.
x,y – The coordinates where the StyledString should be drawn.

See Also:

setPaint, setColor, transform, setTransform, setComposite, clip, setClip, Font

o **drawString**

```
public abstract void drawString(GlyphSet g,
                                float x,
                                float y)
```

Draws a GlyphSet. The rendering attributes applied include the clip, transform, paint or color, and composite attributes. The glyphSet specifies individual glyphs from a Font.

Parameters:

g – The GlyphSet to be drawn.

x,y – The coordinates where the glyphs should be drawn.

See Also:

setPaint, setColor, transform, setTransform, setComposite, clip, setClip

o **drawString**

```
public abstract void drawString(TextLayout text,
                                float x,
                                float y)
```

o **fill**

```
public abstract void fill(Shape s)
```

Fills the interior of a Shape using the settings of the current graphics state. The rendering attributes applied include the clip, transform, paint or color, and composite.

See Also:

setPaint, setColor, transform, setTransform, setComposite, clip, setClip

o **hit**

```
public abstract boolean hit(Rectangle rect,
                            Shape s,
                            boolean onStroke)
```

Checks to see if the outline of a Shape intersects the specified Rectangle in device space. The rendering attributes taken into account include the clip, transform, and stroke attributes.

Parameters:

rect – The area in device space to check for a hit.

s – The shape to check for a hit.

onStroke – Flag to choose between testing the stroked or the filled shape.

Returns:

True if there is a hit, false otherwise.

See Also:

setStroke, fill, draw, transform, setTransform, clip, setClip

o **hitString**

```
public abstract boolean hitString(Rectangle rect,
                                  StyledString s,
                                  float x,
                                  float y)
```

Checks to see if the StyledString intersects the specified Rectangle in device space. The rendering attributes taken into account include the clip and transform.

Parameters:
rect – The area in device space to check for a hit.
s – The StyledString to check for a hit.
x,y – The coordinates where the StyledString should be hit tested.
Returns:
True if there is a hit, false otherwise.
See Also:
drawString, transform, setTransform, clip, setClip

o **getDeviceConfiguration**

```
public abstract GraphicsConfiguration getDeviceConfiguration()
```

Returns the device configuration associated with this Graphics2D.

o **setComposite**

```
public abstract void setComposite(Composite comp)
```

Sets the Composite in the current graphics state. Composite is used in all drawing methods such as drawImage, drawString, draw, and fill. It specifies how new pixels are to be combined with the existing pixels on the graphics device in the rendering process.

Parameters:
comp – The Composite object to be used for drawing.
See Also:
setXORMode, setPaintMode, AlphaComposite

o **setPaint**

```
public abstract void setPaint(Paint paint)
```

Sets the Paint in the current graphics state.

Parameters:
paint – The Paint object to be used to generate color in the rendering process.
See Also:

setColor, GradientPaint, TexturePaint

o **setStroke**

```
public abstract void setStroke(Stroke s)
```

Sets the Stroke in the current graphics state.

Parameters:
s − The Stroke object to be used to stroke a Shape in the rendering process.
See Also:
BasicStroke

o **setRenderingHints**

```
public abstract void setRenderingHints(int hintCategory,
                                       int hintValue)
```

Sets the preferences for the rendering algorithms. Hint categories include controls for rendering quality and overall time/quality trade−off in the rendering process.

Parameters:
hintCategory − The category of hint to be set. Possible values are ANTIALIASING and RENDERING.
hintValue − The value indicating preferences for the specified hint category. Possible values for the ANTIALIASING category are ANTIALIAS_ON, ANTIALIAS_OFF, ANTIALIAS_DEFAULT. Possible values for the RENDERING hint are RENDER_SPEED, RENDER_QUALITY, RENDER_DEFAULT.
See Also:
ANTIALIASING, RENDERING, ANTIALIAS_ON, ANTIALIAS_OFF, ANTIALIAS_DEFAULT, RENDER_SPEED, RENDER_QUALITY, RENDER_DEFAULT

o **getRenderingHints**

```
public abstract int getRenderingHints(int hintCategory)
```

Returns the preferences for the rendering algorithms.

Parameters:
hintCategory − The category of hint to be set. Possible values are ANTIALIASING and RENDERING.
Returns:
The preferences for rendering algorithms. Possible values for the ANTIALIASING category are ANTIALIAS_ON, ANTIALIAS_OFF, ANTIALIAS_DEFAULT. Possible values for the RENDERING hint are RENDER_SPEED, RENDER_QUALITY, RENDER_DEFAULT.
See Also:
ANTIALIASING, RENDERING, ANTIALIAS_ON, ANTIALIAS_OFF,

ANTIALIAS_DEFAULT, RENDER_SPEED, RENDER_QUALITY,
RENDER_DEFAULT

o **transform**

```
public abstract void transform(AffineTransform Tx)
```

Composes a Transform object with the transform in this Graphics2D according to the rule last–specified–first–applied. If the currrent transform is Cx, the result of composition with Tx is a new transform Cx'. Cx' becomes the current transform for this Graphics2D. Transforming a point p by the updated transform Cx' is equivalent to first transforming p by Tx and then transforming the result by the original transform Cx. In other words, Cx'(p) = Cx(Tx(p)). A copy of the Tx is made, if necessary, so further modifications to Tx do not affect rendering.

Parameters:
Tx – The Transform object to be composed with the current transform.
See Also:
setTransform, TransformChain, AffineTransform

o **setTransform**

```
public abstract void setTransform(AffineTransform Tx)
```

Sets the Transform in the current graphics state.

Parameters:
Tx – The Transform object to be used in the rendering process.
See Also:
transform, TransformChain, AffineTransform

o **getTransform**

```
public abstract AffineTransform getTransform()
```

Returns the current Transform in the Graphics2D state.

See Also:
transform, setTransform

o **getPaint**

```
public abstract Paint getPaint()
```

Returns the current Paint in the Graphics2D state.

See Also:
setPaint, setColor

o **getComposite**

```
public abstract Composite getComposite()
```

Returns the current Composite in the Graphics2D state.

See Also:
setComposite

o **setBackground**

```
public abstract void setBackground(Color color)
```

Sets the background color in this context used for clearing a region. When Graphics2D is constructed for a component, the backgroung color is inherited from the component. Setting the background color in the Graphics2D context only affects the subsequent clearRect() calls and not the background color of the component. To change the background of the component, use appropriate methods of the component.

Parameters:
color – The background color that should be used in subsequent calls to clearRect().
See Also:
getBackground, Graphics.clearRect

o **getBackground**

```
public abstract Color getBackground()
```

Returns the background color used for clearing a region.

See Also:
setBackground

o **getStroke**

```
public abstract Stroke getStroke()
```

Returns the current Stroke in the Graphics2D state.

See Also:
setStroke

o **clip**

```
public abstract void clip(Shape s)
```

Intersects the current clip with the interior of the specified Shape and sets the current clip to the resulting intersection. The indicated shape is transformed with the current transform in the

Graphics2D state before being intersected with the current clip. This method is used to make the current clip smaller. To make the clip larger, use any setClip method.

Parameters:

s – The Shape to be intersected with the current clip.

Class java.awt.GraphicsConfiguration

```
java.lang.Object
   |
   +----java.awt.GraphicsConfiguration
```

---

public abstract class **GraphicsConfiguration**
extends Object

This class describes the characteristics of a physical graphics destination such as a printer or monitor. There can be many GraphicsConfiguration objects associated with a single graphics device. For instance, on X11 windowing systems, each visual is a different GraphicsConfiguration. On PCs and Macintoshes, the different screen resolution/color resolution combinations would be different GraphicsConfigurations.

See Also:
GraphicsEnvironment, GraphicsDevice

Constructor Index

o **GraphicsConfiguration**()

Method Index

o **createCompatibleImage**(int, int)
Returns a BufferedImage with channel layout and color model compatible with this graphics configuration.
o **createCompatibleImage**(int, int, int)
Returns a BufferedImage that supports the specified transparency and has a channel layout and color model compatible with this graphics configuration.
o **getColorModel**()
Returns the color model associated with this configuration.
o **getColorModel**(int)
Returns the color model associated with this configuration that supports the specified transparency.
o **getDefaultTransform**()

Returns the default Transform for this configuration.

o **getDevice**()

Return the graphics device associated with this configuration.

o **getNormalizingTransform**()

Returns a Transform that can be composed with the default Transform of a Graphics2D so that 72 units in user space will equal 1 inch in device space.

Constructors

o **GraphicsConfiguration**

```
public GraphicsConfiguration()
```

Methods

o **getDevice**

```
public abstract GraphicsDevice getDevice()
```

Return the graphics device associated with this configuration.

o **createCompatibleImage**

```
public abstract BufferedImage createCompatibleImage(int width,
                                                    int height)
```

Returns a BufferedImage with channel layout and color model compatible with this graphics configuration. This method has nothing to do with memory−mapping a device. This BufferedImage has a layout and color model that is closest to this native device configuration and thus can be optimally blitted to this device.

o **createCompatibleImage**

```
public abstract BufferedImage createCompatibleImage(int width,
                                                    int height,
                                                    int transparency)
```

Returns a BufferedImage that supports the specified transparency and has a channel layout and color model compatible with this graphics configuration. This method has nothing to do with memory−mapping a device. This BufferedImage has a layout and color model that can be optimally blitted to a device with this configuration.

See Also:

OPAQUE, BITMASK, TRANSLUCENT

o **getColorModel**

```
public abstract ColorModel getColorModel()
```

Returns the color model associated with this configuration.

o **getColorModel**

```
public abstract ColorModel getColorModel(int transparency)
```

Returns the color model associated with this configuration that supports the specified transparency.

o **getDefaultTransform**

```
public abstract AffineTransform getDefaultTransform()
```

Returns the default Transform for this configuration. This Transform is typically the Identity transform for most normal screens. Device coordinates for screen and printer devices will have the origin in the upper left–hand corner of the target region of the device, with X coordinates increasing to the right and Y coordinates increasing downwards. For image buffers, this Transform will be the Identity transform.

o **getNormalizingTransform**

```
public abstract AffineTransform getNormalizingTransform()
```

Returns a Transform that can be composed with the default Transform of a Graphics2D so that 72 units in user space will equal 1 inch in device space. Given a Graphics2D, g, one can reset the transformation to create such a mapping by using the following pseudocode:

```
GraphicsConfiguration gc = g.getGraphicsConfiguration();
g.setTransform(gc.getDefaultTransform());
g.transform(gc.getNormalizingTransform());
```

Note that sometimes this Transform will be identity (e.g. for printers or metafile output) and that this Transform is only as accurate as the information supplied by the underlying system. For image buffers, this Transform will be the Identity transform, since there is no valid distance measurement.

Class java.awt.GraphicsDevice

```
java.lang.Object
   |
   +----java.awt.GraphicsDevice
```

---

public abstract class **GraphicsDevice**
extends Object

This class describes the graphics devices that may be available in a graphics environment. These include screen and printer devices. Note that there may be many screens and many printers in an instance of GraphicsEnvironment. Each graphics device has one or more GraphicsConfiguration objects associated with it. These specify the different configurations in which the GraphicsDevice can be used.

See Also:
GraphicsEnvironment, GraphicsConfiguration

Variable Index

o **TYPE_IMAGE_BUFFER**
Device is an image buffer.
o **TYPE_PRINTER**
Device is a printer.
o **TYPE_RASTER_SCREEN**
Device is a raster screen.

Constructor Index

o **GraphicsDevice**()

Method Index

o **getConfigurations**()
Returns all of the graphics configurations associated with this graphics device.
o **getDefaultConfiguration**()

Returns the default graphics configuration associated with this graphics device.

o **getIDstring**()

Returns the identification string associated with this graphics device.

o **getType**()

Returns the type of the graphics device.

Variables

o TYPE_RASTER_SCREEN

```
public static final int TYPE_RASTER_SCREEN
```

Device is a raster screen.

o TYPE_PRINTER

```
public static final int TYPE_PRINTER
```

Device is a printer.

o TYPE_IMAGE_BUFFER

```
public static final int TYPE_IMAGE_BUFFER
```

Device is an image buffer. This buffer may reside in device or system memory but it is not visible to the user.

Constructors

o GraphicsDevice

```
public GraphicsDevice()
```

Methods

o getType

```
public abstract int getType()
```

Returns the type of the graphics device.

See Also:

TYPE_RASTER_SCREEN, TYPE_PRINTER, TYPE_IMAGE_BUFFER

o getIDstring

```
public abstract String getIDstring()
```

Returns the identification string associated with this graphics device.

o **getConfigurations**

```
public abstract GraphicsConfiguration[] getConfigurations()
```

Returns all of the graphics configurations associated with this graphics device.

o **getDefaultConfiguration**

```
public abstract GraphicsConfiguration getDefaultConfiguration()
```

Returns the default graphics configuration associated with this graphics device.

Class java.awt.GraphicsEnvironment

```
java.lang.Object
   |
   +----java.awt.GraphicsEnvironment
```

---

public abstract class **GraphicsEnvironment**
extends Object

This class specifies the graphics environment. The resources in this environment might be local or on a remote machine. The graphics environment consists of a number of GraphicsDevice objects and Font objects. GraphicsDevice objects are typically screens or printers and are the destination of Graphics2D drawing methods. Each GraphicsDevice has a number of GraphicsConfiguration objects associated with it. These specify the different configurations in which the GraphicsDevice can be used.

See Also:
GraphicsDevice, GraphicsConfiguration

Constructor Index

o **GraphicsEnvironment**()

Method Index

o **createGraphics**(BufferedImage)
Returns a Graphics2D object for rendering into the given BufferedImage.
o **getAllFonts**()
Returns all fonts available in this environment.
o **getDefaultScreenDevice**()
Returns the default screen graphics device.
o **getFont**(String)
o **getFonts**(TextAttributeSet)
Returns all Fonts available in this environment that match the specified AttributeSet.
o **getImagingLib**()
Returns a ImagingLib object that can used to process some/all of the imaging ops.

o **getLocalGraphicsEnvironment**()
Returns the local graphics environment.
o **getScreenDevices**()
Returns an array of all of the screen devices.

Constructors

o GraphicsEnvironment

```
protected GraphicsEnvironment()
```

Methods

o getLocalGraphicsEnvironment

```
public static GraphicsEnvironment getLocalGraphicsEnvironment()
```

Returns the local graphics environment.

o getScreenDevices

```
public abstract GraphicsDevice[] getScreenDevices()
```

Returns an array of all of the screen devices.

o getDefaultScreenDevice

```
public abstract GraphicsDevice getDefaultScreenDevice()
```

Returns the default screen graphics device.

o createGraphics

```
public abstract Graphics2D createGraphics(BufferedImage img)
```

Returns a Graphics2D object for rendering into the given BufferedImage.

o getImagingLib

```
public abstract ImagingLib getImagingLib()
```

Returns a ImagingLib object that can used to process some/all of the imaging ops. Can be null.

o getAllFonts

```
public abstract Font[] getAllFonts()
```

Returns all fonts available in this environment.

o **getFonts**

```
public abstract Font[] getFonts(TextAttributeSet attributes)
```

Returns all Fonts available in this environment that match the specified AttributeSet.

o **getFont**

```
public abstract Font getFont(String fontFaceName)
```

Class java.awt.Rectangle

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

---

public class **Rectangle**
extends Rectangle2D
implements Shape, Serializable

A rectangle specifies an area in a coordinate space that is defined by the rectangle's top–left point (*x*, *y*) in the coordinate space, its width, and its height.

A rectangle's `width` and `height` are public fields. The constructors that allow you to create a rectangle, and the methods that allow you to modify one, do not prevent you from setting a negative value for width or height.

A rectangle whose width or height is negative is considered empty, and all methods defined by the `Rectangle` class behave accordingly. If the rectangle is empty, then the method `isEmpty` returns `true`. No point can be contained by or inside an empty rectangle, however the values of `width` and `height` are still valid. An empty rectangle still has a location in the coordinate space, and methods that change its size or location remain valid. The behavior of methods that operate on more than one rectangle is undefined if any of the participating rectangles has a negative `width` or `height`. These methods include `intersects`, `intersection`, and `union`. specified in integer coordinates.

Variable Index

o **height**
    The height of the rectangle.
o **width**
    The width of the rectangle.
o **x**
    The *x* coordinate of the rectangle.
o **y**

The *y* coordinate of the rectangle.

Constructor Index

o **Rectangle**()
  Constructs a new rectangle whose top−left corner is at (0, 0) in the coordinate space, and whose width and height are zero.
o **Rectangle**(Dimension)
  Constructs a new rectangle whose top left corner is (0, 0) and whose width and height are specified by the `dimension` argument.
o **Rectangle**(int, int)
  Constructs a new rectangle whose top−left corner is at (0, 0) in the coordinate space, and whose width and height are specified by the arguments of the same name.
o **Rectangle**(int, int, int, int)
  Constructs a new rectangle whose top−left corner is specified as (`x`, `y`) and whose width and height are specified by the arguments of the same name.
o **Rectangle**(Point)
  Constructs a new rectangle whose top−left corner is the specified point, and whose width and height are zero.
o **Rectangle**(Point, Dimension)
  Constructs a new rectangle whose top−left corner is specified by the `point` argument, and whose width and height are specified by the `dimension` argument.
o **Rectangle**(Rectangle)
  Constructs a new rectangle, initialized to match the values of the specificed rectangle.

Method Index

o **add**(int, int)
  Adds a point, specified by the integer arguments `newx` and `newy`, to this rectangle.
o **add**(Point)
  Adds the point `pt` to this rectangle.
o **add**(Rectangle)
  Adds a rectangle to this rectangle.
o **contains**(int, int)
  Checks whether this rectangle contains the point at the specified location (*x*, *y*).
o **contains**(int, int, int, int)
  Checks whether this rectangle entirely contains the rectangle at the specified location (*X*, *Y*) with the specified dimensions (*W*, *H*).
o **contains**(Point)
  Checks whether this rectangle contains the specified point.
o **contains**(Rectangle)
  Checks whether this rectangle entirely contains the specified rectangle.
o **equals**(Object)
  Checks whether two rectangles are equal.
o **getBounds**()
  Gets the bounding rectangle of this rectangle.
o **getHeight**()

Returns the height of the bounding rectangle in double precision.

o **getLocation**()
: Returns the location of this rectangle.

o **getSize**()
: Gets the size (width and height) of this rectangle.

o **getWidth**()
: Returns the width of the bounding rectangle in double precision.

o **getX**()
: Returns the X coordinate of the bounding rectangle in double precision.

o **getY**()
: Returns the Y coordinate of the bounding rectangle in double precision.

o **grow**(int, int)
: Grows the rectangle both horizontally and vertically.

o **hashCode**()
: Returns the hashcode for this rectangle.

o **inside**(int, int)
: **Deprecated.**

o **intersection**(Rectangle)
: Computes the intersection of this rectangle with the specified rectangle.

o **intersects**(Rectangle)
: Determines whether this rectangle and the specified rectangle intersect.

o **isEmpty**()
: Determines whether this rectangle is empty.

o **move**(int, int)
: **Deprecated.**

o **outcode**(double, double)
: Determines where the specified float coordinates lie with respect to this Rectangle.

o **reshape**(int, int, int, int)
: **Deprecated.**

o **resize**(int, int)
: **Deprecated.**

o **setBounds**(int, int, int, int)
: Sets the bounding rectangle of this rectangle to the specified values for `x`, `y`, `width`, and `height`.

o **setBounds**(Rectangle)
: Sets the bounding rectangle of this rectangle to match the specified rectangle.

o **setLocation**(int, int)
: Moves the rectangle to the specified location.

o **setLocation**(Point)
: Moves the rectangle to the specified location.

o **setRect**(double, double, double, double)
: Set the bounds of this rectangle to the specified x, y, width, and height.

o **setSize**(Dimension)
: Sets the size of this rectangle to match the specified dimension.

o **setSize**(int, int)
: Sets the size of this rectangle to the specified width and height.

o **toString**()
: Returns a string representation of this rectangle and its values.

o **translate**(int, int)

Translates the rectangle the indicated distance, to the right along the *x* coordinate axis, and downward along the *y* coordinate axis.

o **union**(Rectangle)

Computes the union of this rectangle with the specified rectangle.

Variables

o **x**

```
public int x
```

The *x* coordinate of the rectangle.

o **y**

```
public int y
```

The *y* coordinate of the rectangle.

o **width**

```
public int width
```

The width of the rectangle.

o **height**

```
public int height
```

The height of the rectangle.

Constructors

o **Rectangle**

```
public Rectangle()
```

Constructs a new rectangle whose top–left corner is at (0, 0) in the coordinate space, and whose width and height are zero.

o **Rectangle**

```
public Rectangle(Rectangle r)
```

Constructs a new rectangle, initialized to match the values of the specificed rectangle.

Parameters:

r – a rectangle from which to copy initial values.

o **Rectangle**

```
public Rectangle(int x,
                 int y,
                 int width,
                 int height)
```

Constructs a new rectangle whose top–left corner is specified as (`x`, `y`) and whose width and height are specified by the arguments of the same name.

Parameters:
x – the *x* coordinate.
y – the *y* coordinate.
width – the width of the rectangle.
height – the height of the rectangle.

o **Rectangle**

```
public Rectangle(int width,
                 int height)
```

Constructs a new rectangle whose top–left corner is at (0, 0) in the coordinate space, and whose width and height are specified by the arguments of the same name.

Parameters:
width – the width of the rectangle.
height – the height of the rectangle.

o **Rectangle**

```
public Rectangle(Point p,
                 Dimension d)
```

Constructs a new rectangle whose top–left corner is specified by the `point` argument, and whose width and height are specified by the `dimension` argument.

Parameters:
p – a point, the top–left corner of the rectangle.
d – a dimension, representing the width and height.

o **Rectangle**

```
public Rectangle(Point p)
```

Constructs a new rectangle whose top–left corner is the specified point, and whose width and height are zero.

Parameters:
p – the top left corner of the rectangle.

o **Rectangle**

```
public Rectangle(Dimension d)
```

Constructs a new rectangle whose top left corner is (0, 0) and whose width and height are specified by the `dimension` argument.

Parameters:
d – a dimension, specifying width and height.

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the bounding rectangle in double precision.

Overrides:
getX in class RectangularShape

o **getY**

```
public double getY()
```

Returns the Y coordinate of the bounding rectangle in double precision.

Overrides:
getY in class RectangularShape

o **getWidth**

```
public double getWidth()
```

Returns the width of the bounding rectangle in double precision.

Overrides:
getWidth in class RectangularShape

o **getHeight**

```
public double getHeight()
```

Returns the height of the bounding rectangle in double precision.

Overrides:
getHeight in class RectangularShape

o **getBounds**

```
public Rectangle getBounds()
```

Gets the bounding rectangle of this rectangle.

This method is included for completeness, to parallel the `getBounds` method of `Component` .

Returns:
a new rectangle, equal to the bounding rectangle for this rectangle.
Overrides:
getBounds in class RectangularShape
See Also:
getBounds

o **setBounds**

```
public void setBounds(Rectangle r)
```

Sets the bounding rectangle of this rectangle to match the specified rectangle.

This method is included for completeness, to parallel the `setBounds` method of `Component` .

Parameters:
r – a rectangle.
See Also:
setBounds

o **setBounds**

```
public void setBounds(int x,
                      int y,
                      int width,
                      int height)
```

Sets the bounding rectangle of this rectangle to the specified values for `x`, `y`, `width`, and `height`.

This method is included for completeness, to parallel the `setBounds` method of `Component` .

Parameters:
x – the new *x* coordinate for the top–left corner of this rectangle.
y – the new *y* coordinate for the top–left corner of this rectangle.

width – the new width for this rectangle.
height – the new height for this rectangle.

See Also:

setBounds

o **setRect**

```
public void setRect(double x,
                    double y,
                    double width,
                    double height)
```

Set the bounds of this rectangle to the specified x, y, width, and height. This method is included for completeness, to parallel the setBounds method of Component.

Parameters:

width – the new width for the Dimension object
height – the new height for the Dimension object

Overrides:

setRect in class Rectangle2D

o **reshape**

```
public void reshape(int x,
                    int y,
                    int width,
                    int height)
```

Note: reshape() is deprecated. *As of JDK version 1.1, replaced by* `setBounds(int, int, int, int)`.

o **getLocation**

```
public Point getLocation()
```

Returns the location of this rectangle.

This method is included for completeness, to parallel the `getLocation` method of `Component`.

See Also:

getLocation

o **setLocation**

```
public void setLocation(Point p)
```

Moves the rectangle to the specified location.

This method is included for completeness, to parallel the `setLocation` method of `Component`.

Parameters:
p – the new location for the point.
See Also:
setLocation

o **setLocation**

```
public void setLocation(int x,
                        int y)
```

Moves the rectangle to the specified location.

This method is included for completeness, to parallel the `setLocation` method of `Component`.

Parameters:
x – the *x* coordinate of the new location.
y – the *y* coordinate of the new location.
See Also:
setLocation

o **move**

```
public void move(int x,
                 int y)
```

Note: move() is deprecated. *As of JDK version 1.1, replaced by* `setLocation(int, int)`.

o **translate**

```
public void translate(int x,
                      int y)
```

Translates the rectangle the indicated distance, to the right along the *x* coordinate axis, and downward along the *y* coordinate axis.

Parameters:
dx – the distance to move the rectangle along the *x* axis.
dy – the distance to move the rectangle along the *y* axis.
See Also:
setLocation, setLocation

o **getSize**

```
public Dimension getSize()
```

Gets the size (width and height) of this rectangle.

This method is included for completeness, to parallel the `getSize` method of `Component`.

Returns:
a dimension, representing the size.

See Also:
getSize

o **setSize**

```
public void setSize(Dimension d)
```

Sets the size of this rectangle to match the specified dimension.

This method is included for completeness, to parallel the `setSize` method of `Component`.

Parameters:
d – the new size for the Dimension object

See Also:
setSize

o **setSize**

```
public void setSize(int width,
                    int height)
```

Sets the size of this rectangle to the specified width and height.

This method is included for completeness, to parallel the `setSize` method of `Component`.

Parameters:
width – the new width for this rectangle object.
height – the new height for this rectangle object.

See Also:
setSize

o **resize**

```
public void resize(int width,
                   int height)
```

Note: resize() is deprecated. *As of JDK version 1.1, replaced by* `setSize(int, int)`.

o **contains**

```
public boolean contains(Point p)
```

Checks whether this rectangle contains the specified point.

Parameters:
p – the point (location) to test.
Returns:
`true` if the point (*x*, *y*) is inside this rectangle; `false` otherwise.

o **contains**

```
public boolean contains(int x,
                        int y)
```

Checks whether this rectangle contains the point at the specified location (*x*, *y*).

Parameters:
x – the *x* coordinate.
y – the *y* coordinate.
Returns:
`true` if the point (*x*, *y*) is inside this rectangle; `false` otherwise.

o **contains**

```
public boolean contains(Rectangle r)
```

Checks whether this rectangle entirely contains the specified rectangle.

Parameters:
r – the rectangle.
Returns:
`true` if the rectangle (*r*) is contained entirely inside this rectangle; `false` otherwise.

o **contains**

```
public boolean contains(int X,
                        int Y,
                        int W,
                        int H)
```

Checks whether this rectangle entirely contains the rectangle at the specified location (*X*, *Y*) with the specified dimensions (*W*, *H*).

Parameters:
X – the *x* coordinate.
Y – the *y* coordinate.
W – the *width* of the rectangle.
H – the *height* of the rectangle.
Returns:

`true` if the rectangle specified by (*X*, *Y*, *W*, *H*) is entirely enclosed inside this rectangle; `false` otherwise.

o **inside**

```
public boolean inside(int x,
                      int y)
```

Note: inside() is deprecated.*As of JDK version 1.1, replaced by* `contains(int, int)`.

o **intersects**

```
public boolean intersects(Rectangle r)
```

Determines whether this rectangle and the specified rectangle intersect. Two rectangles intersect if their intersection is nonempty.

Parameters:
r – a rectangle.
Returns:
`true` if the specified rectangle and this rectangle insersect; `false` otherwise.

o **intersection**

```
public Rectangle intersection(Rectangle r)
```

Computes the intersection of this rectangle with the specified rectangle. Returns a new rectangle that represents the intersection of the two rectangles.

Parameters:
r – a rectangle.
Returns:
the largest rectangle contained in both the specified rectangle and in this rectangle.

o **union**

```
public Rectangle union(Rectangle r)
```

Computes the union of this rectangle with the specified rectangle. Returns a new rectangle that represents the union of the two rectangles.

Parameters:
r – a rectangle.
Returns:
the smallest rectangle containing both the specified rectangle and this rectangle.

o **add**

```
public void add(int newx,
```

```
                int newy)
```

Adds a point, specified by the integer arguments `newx` and `newy`, to this rectangle. The resulting rectangle is the smallest rectangle that contains both the original rectangle and the specified point.

After adding a point, a call to `contains with the added point as an argument will not necessarily return true. The contains method does not return true for points on the right or bottom edges of a rectangle. Therefore if the added point falls on the left or bottom edge of the enlarged rectangle, contains will return false for that point.`

Parameters:

`newx - the` *x* `coordinate of the new point.`
`newy - the` *y* `coordinate of the new point.`

o add

```
public void add(Point pt)
```

Adds the point `pt` to this rectangle. The resulting rectangle is the smallest rectangle that contains both the original rectangle and the specified point.

After adding a point, a call to `contains with the added point as an argument will not necessarily return true. The contains method does not return true for points on the right or bottom edges of a rectangle. Therefore if the added point falls on the left or bottom edge of the enlarged rectangle, contains will return false for that point.`

Parameters:

`pt - the new point to add to the rectangle.`

o add

```
public void add(Rectangle r)
```

Adds a rectangle to this rectangle. The resulting rectangle is the union of the two rectangles.

Parameters:

a – rectangle.

o grow

```
public void grow(int h,
                 int v)
```

Grows the rectangle both horizontally and vertically.

This method modifies the rectangle so that it is `h` units larger on both the left and right side, and `v` units larger at both the top and bottom.

The new rectangle has (`x - h`, `y - v`) as its top–left corner, a width of `width` + `2h`, and a height of `height` + `2v`.

If negative values are supplied for `h` and `v`, the size of the rectangle decreases accordingly. The `grow` method does not check whether the resulting values of `width` and `height` are non–negative.

Parameters:
h – the horizontal expansion.
v – the vertical expansion.

o **isEmpty**

```
public boolean isEmpty()
```

Determines whether this rectangle is empty. A rectangle is empty if its width or its height is less than or equal to zero.

Returns:
`true` if this rectangle is empty; `false` otherwise.
Overrides:
isEmpty in class RectangularShape

o **outcode**

```
public int outcode(double x,
                   double y)
```

Determines where the specified float coordinates lie with respect to this Rectangle. This method computes a binary OR of the appropriate mask values indicating which sides of the rectangle the given point is outside of.

Returns:
the logical OR of all appropriate out codes
Overrides:
outcode in class Rectangle2D
See Also:
OUT_LEFT, OUT_TOP, OUT_RIGHT, OUT_BOTTOM

o **hashCode**

```
public int hashCode()
```

Returns the hashcode for this rectangle.

Returns:
the hashcode for this rectangle.
Overrides:
hashCode in class Object

o **equals**

```
public boolean equals(Object obj)
```

Checks whether two rectangles are equal.

The result is **`true`** if and only if the argument is not **`null`** and is a **`Rectangle`** object that has the same top–left corner, width, and height as this rectangle.

Parameters:
obj – the object to compare with.
Returns:
`true` if the objects are equal; `false` otherwise.
Overrides:
equals in class Object

o **toString**

```
public String toString()
```

Returns a string representation of this rectangle and its values.

Returns:
a string representation of this rectangle.
Overrides:
toString in class Object

Class java.awt.TexturePaint

```
java.lang.Object
   |
   +----java.awt.TexturePaint
```

public class **TexturePaint**
extends Object
implements Paint

This class provides a way to fill a shape with a given texture. The texture used in the filling process is given as a BufferedImage. The size of the texture BufferedImage object should be small because the BufferedImage data is copied by the TexturePaint object. The texture is anchored at construction time to the upper left corner of a Rectangle2D specified in user space. Texture is computed for locations in the device space by conceptually replicating the given Rectangle2D infinitely in all directions in user space and mapping the BufferedImage to each replicated Rectangle2D.

See Also:
Paint, setPaint

Variable Index

o **BICUBIC**
Texture colors are generated with the bicubic algorithm.
o **BILINEAR**
Texture colors are generated with the bilinear algorithm.
o **NEAREST_NEIGHBOR**
Texture colors are generated with the nearest neighbor algorithm.

Constructor Index

o **TexturePaint**(BufferedImage, Rectangle2D, int)
Constructs a TexturePaint object.

Method Index

o **createContext**(ColorModel, Rectangle, Rectangle2D, AffineTransform)
Creates and returns a context used to generate the color pattern.
o **getTransparency**()
Return the transparency mode for this TexturePaint.

Variables

o **NEAREST_NEIGHBOR**

```
public static final int NEAREST_NEIGHBOR
```

Texture colors are generated with the nearest neighbor algorithm.

o **BILINEAR**

```
public static final int BILINEAR
```

Texture colors are generated with the bilinear algorithm.

o **BICUBIC**

```
public static final int BICUBIC
```

Texture colors are generated with the bicubic algorithm.

Constructors

o **TexturePaint**

```
public TexturePaint(BufferedImage txtr,
                    Rectangle2D rect2d,
                    int interpolation)
```

Constructs a TexturePaint object.

Parameters:
txtr – The BufferedImage object with the texture used for painting.
rect2d – The rectangle in user space used to anchor and replicate the texture.
interpolation – The type of algorithm with regard to rendering quality used to generate the color pattern. Possible values are NEAREST_NEIGHBOR, BILINEAR and BICUBIC.

See Also:
NEAREST_NEIGHBOR, BILINEAR, BICUBIC

Methods

o **createContext**

```
public PaintContext createContext(ColorModel cm,
                                  Rectangle deviceBounds,
                                  Rectangle2D userBounds,
                                  AffineTransform xform)
```

Creates and returns a context used to generate the color pattern.

Parameters:
cm – ColorModel in which the caller wishes to receive the paint data. This is used only as a hint.
deviceBounds – The rectangle describing the bounding box in device space of the graphics primitive being rendered.
userBounds – The rectangle describing the bounding box in user space of the graphics primitive being rendered.
xform – The Transform from user space into device space.

Returns:
The PaintContext for generating color patterns.

See Also:
PaintContext

o **getTransparency**

```
public int getTransparency()
```

Return the transparency mode for this TexturePaint.

See Also:
Transparency

Class java.awt.color.ColorSpace

```
java.lang.Object
   |
   +----java.awt.color.ColorSpace
```

public abstract class **ColorSpace**
extends Object
super class of:
ICC_ColorSpace

This abstract class is used to serve as a color space tag to identify the specific color space of a Color object or, via a ColorModel object, of an Image, a BufferedImage, or a GraphicsDevice. It contains methods that transform Colors in a specific color space to/from sRGB and to/from a well−defined CIEXYZ color space.

Several variables are defined for purposes of referring to color space types (e.g. TYPE_RGB, TYPE_XYZ, etc.) and to refer to specific color spaces (e.g. CS_sRGB and CS_CIEXYZ). sRGB is a proposed standard RGB color space. For more information, see http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html .

The purpose of the methods to transform to/from the well−defined CIEXYZ color space is to support conversions between any two color spaces at a reasonably high degree of accuracy. It is expected that particular implementations of subclasses of ColorSpace (e.g. ICC_ColorSpace) will support high performance conversion based on underlying platform color management systems.

The CS_CIEXYZ space used by the toCIEXYZ/fromCIEXYZ methods can be described as follows:

```
CIEXYZ
viewing illuminance: 200 lux
viewing white point: CIE D50
media white point: "that of a perfectly reflecting diffuser" -- D50
media black point: 0 lux or 0 Reflectance
flare: 1 percent
surround: 20percent of the media white point
media description: reflection print (i.e., RLAB, Hunt viewing media)
note: For developers creating an ICC profile for this conversion space,
the following is applicable.  Use a simple Von Kries white point
adaptation folded into the 3X3 matrix parameters and fold the
flare and surround effects into the three one-dimensional
lookup tables (assuming one uses the minimal model for
monitors).
```

See Also:

ICC_ColorSpace

Variable Index

o **CS_CIEXYZ**
The CIEXYZ conversion color space defined above.
o **CS_sRGB**
The sRGB color space defined at http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html .
o **TYPE_CMY**
Any of the family of CMY color spaces.
o **TYPE_CMYK**
Any of the family of CMYK color spaces.
o **TYPE_GRAY**
Any of the family of GRAY color spaces.
o **TYPE_HLS**
Any of the family of HLS color spaces.
o **TYPE_HSB**
Any of the family of HSB color spaces.
o **TYPE_HSV**
Any of the family of HSV color spaces.
o **TYPE_Lab**
Any of the family of Lab color spaces.
o **TYPE_Luv**
Any of the family of Luv color spaces.
o **TYPE_RGB**
Any of the family of RGB color spaces.
o **TYPE_XYZ**
Any of the family of XYZ color spaces.
o **TYPE_YCbCr**
Any of the family of YCbCr color spaces.
o **TYPE_Yxy**
Any of the family of Yxy color spaces.

Constructor Index

o **ColorSpace**(int, int)
Constructs a ColorSpace object given a color space type and the number of components.

Method Index

o **fromCIEXYZ**(float[])
Transforms a color value assumed to be in the CS_CIEXYZ conversion color space into this ColorSpace.
o **fromRGB**(float[])

Transforms a color value assumed to be in the default CS_sRGB color space into this ColorSpace.

o **getInstance**(int)
   Returns a ColorSpace representing one of the specific predefined color spaces.

o **getName**(int)
   Returns the name of the component given the component index

o **getNumComponents**()
   Returns the number of components of this ColorSpace.

o **getType**()
   Returns the color space type of this ColorSpace (for example TYPE_RGB, TYPE_XYZ, ...).

o **isCS_sRGB**()
   Returns true if the ColorSpace is CS_sRGB

o **toCIEXYZ**(float[])
   Transforms a color value assumed to be in this ColorSpace into the CS_CIEXYZ conversion color space.

o **toRGB**(float[])
   Transforms a color value assumed to be in this ColorSpace into a value in the default CS_sRGB color space.

Variables

o **TYPE_XYZ**

```
public static final int TYPE_XYZ
```

Any of the family of XYZ color spaces.

o **TYPE_Lab**

```
public static final int TYPE_Lab
```

Any of the family of Lab color spaces.

o **TYPE_Luv**

```
public static final int TYPE_Luv
```

Any of the family of Luv color spaces.

o **TYPE_YCbCr**

```
public static final int TYPE_YCbCr
```

Any of the family of YCbCr color spaces.

o **TYPE_Yxy**

```
public static final int TYPE_Yxy
```

Any of the family of Yxy color spaces.

o **TYPE_RGB**

```
public static final int TYPE_RGB
```

Any of the family of RGB color spaces.

o **TYPE_GRAY**

```
public static final int TYPE_GRAY
```

Any of the family of GRAY color spaces.

o **TYPE_HSV**

```
public static final int TYPE_HSV
```

Any of the family of HSV color spaces.

o **TYPE_HLS**

```
public static final int TYPE_HLS
```

Any of the family of HLS color spaces.

o **TYPE_CMYK**

```
public static final int TYPE_CMYK
```

Any of the family of CMYK color spaces.

o **TYPE_CMY**

```
public static final int TYPE_CMY
```

Any of the family of CMY color spaces.

o **TYPE_HSB**

```
public static final int TYPE_HSB
```

Any of the family of HSB color spaces.

o **CS_sRGB**

```
public static final int CS_sRGB
```

The sRGB color space defined at http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html .

o **CS_CIEXYZ**

```
public static final int CS_CIEXYZ
```

The CIEXYZ conversion color space defined above.

Constructors

o **ColorSpace**

```
public ColorSpace(int type,
                  int numcomponents)
```

Constructs a ColorSpace object given a color space type and the number of components.

Methods

o **getInstance**

```
public static ColorSpace getInstance(int colorspace)
```

Returns a ColorSpace representing one of the specific predefined color spaces.

Parameters:
colorspace – a specific color space identified by one of the predefined class constants (e.g. CS_sRGB or CS_CIEXYZ)

o **isCS_sRGB**

```
public boolean isCS_sRGB()
```

Returns true if the ColorSpace is CS_sRGB

o **toRGB**

```
public abstract float[] toRGB(float colorvalue[])
```

Transforms a color value assumed to be in this ColorSpace into a value in the default CS_sRGB color space.

Parameters:
colorvalue – a float array with length of at least the number of components in this ColorSpace

Returns:
a float array of length 3

o **fromRGB**

```
public abstract float[] fromRGB(float rgbvalue[])
```

Transforms a color value assumed to be in the default CS_sRGB color space into this ColorSpace.

Parameters:
rgbvalue – a float array with length of at least 3

Returns:
a float array with length equal to the number of components in this ColorSpace

o **toCIEXYZ**

```
public abstract float[] toCIEXYZ(float colorvalue[])
```

Transforms a color value assumed to be in this ColorSpace into the CS_CIEXYZ conversion color space.

Parameters:
colorvalue – a float array with length of at least the number of components in this ColorSpace

Returns:
a float array of length 3

o **fromCIEXYZ**

```
public abstract float[] fromCIEXYZ(float colorvalue[])
```

Transforms a color value assumed to be in the CS_CIEXYZ conversion color space into this ColorSpace.

Parameters:
colorvalue – a float array with length of at least 3

Returns:
a float array with length equal to the number of components in this ColorSpace

o **getType**

```
public int getType()
```

Returns the color space type of this ColorSpace (for example TYPE_RGB, TYPE_XYZ, ...). The type defines the number of components of the color space and the interpretation, e.g. TYPE_RGB identifies a color space with three components – red, green, and blue. It does not define the particular color characteristics of the space, e.g. the chromaticities of the primaries.

o **getNumComponents**

```
public int getNumComponents()
```

Returns the number of components of this ColorSpace.

o **getName**

```
public String getName(int idx)
```

Returns the name of the component given the component index

Class java.awt.color.ICC_ColorSpace

```
java.lang.Object
   |
   +----java.awt.color.ColorSpace
           |
           +----java.awt.color.ICC_ColorSpace
```

public class **ICC_ColorSpace**
extends ColorSpace

An implementation of the abstract ColorSpace class. This representation of device independent and device dependent color spaces is based on the ICC Profile Format Specification, Version 3.3, November 11, 1996, from the International Color Consortium (see http://www.color.org).

Typically, a Color or ColorModel would be associated with an ICC Profile which is either an input, display, or output profile (see the ICC specification). There are other types of ICC Profiles, e.g. abstract profiles, device link profiles, and named color profiles, which do not contain information appropriate for representing the color space of a color, image, or device (see ICC_Profile). Attempting to create an ICC_ColorSpace object from an inappropriate ICC Profile is an error.

ICC Profiles represent transformations from the color space of the profile (e.g. a monitor) to a Profile Connection Space (PCS). Profiles of interest for tagging images or colors have a PCS which is one of the two specific device independent spaces (one CIEXYZ space and one CIELab space) defined in the ICC Profile Format Specification. Most profiles of interest either have invertible transformations or explicitly specify transformations going both directions. Should an ICC_ColorSpace object be used in a way requiring a conversion from PCS to the profile's native space and there is inadequate data to correctly perform the conversion, the ICC_ColorSpace object will produce output in the specified type of color space (e.g. TYPE_RGB, TYPE_CMYK, etc.), but the specific color values of the output data will be undefined.

The details of this class are not important for simple applets, which draw in a default color space or manipulate and display imported images with a known color space. At most, such applets would need to get one of the default color spaces via ColorSpace.getInstance().

See Also:
ColorSpace, ICC_Profile

Constructor Index

o **ICC_ColorSpace**(ICC_Profile)
Constructs a new ICC_ColorSpace from an ICC_Profile object.

Method Index

o **fromCIEXYZ**(float[])
Transforms a color value assumed to be in the CS_CIEXYZ conversion color space into this ColorSpace.
o **fromRGB**(float[])
Transforms a color value assumed to be in the default CS_sRGB color space into this ColorSpace.
o **getProfile**()
Returns the ICC_Profile for this ICC_ColorSpace.
o **toCIEXYZ**(float[])
Transforms a color value assumed to be in this ColorSpace into the CS_CIEXYZ conversion color space.
o **toRGB**(float[])
Transforms a color value assumed to be in this ColorSpace into a value in the default CS_sRGB color space.

Constructors

o ICC_ColorSpace

```
public ICC_ColorSpace(ICC_Profile profile)
```

Constructs a new ICC_ColorSpace from an ICC_Profile object.

Throws: IllegalArgumentException
if profile is inappropriate for representing a ColorSpace.

Methods

o getProfile

```
public ICC_Profile getProfile()
```

Returns the ICC_Profile for this ICC_ColorSpace.

o toRGB

```
public float[] toRGB(float colorvalue[])
```

Transforms a color value assumed to be in this ColorSpace into a value in the default CS_sRGB color space.

Parameters:
colorvalue – a float array with length of at least the number of components in this ColorSpace.
Returns:
a float array of length 3.
Overrides:
toRGB in class ColorSpace

o **fromRGB**

```
public float[] fromRGB(float rgbvalue[])
```

Transforms a color value assumed to be in the default CS_sRGB color space into this ColorSpace.

Parameters:
rgbvalue – a float array with length of at least 3.
Returns:
a float array with length equal to the number of components in this ColorSpace.
Overrides:
fromRGB in class ColorSpace

o **toCIEXYZ**

```
public float[] toCIEXYZ(float colorvalue[])
```

Transforms a color value assumed to be in this ColorSpace into the CS_CIEXYZ conversion color space.

Parameters:
colorvalue – a float array with length of at least the number of components in this ColorSpace.
Returns:
a float array of length 3.
Overrides:
toCIEXYZ in class ColorSpace

o **fromCIEXYZ**

```
public float[] fromCIEXYZ(float colorvalue[])
```

Transforms a color value assumed to be in the CS_CIEXYZ conversion color space into this ColorSpace.

Parameters:
colorvalue – a float array with length of at least 3.

Returns:

a float array with length equal to the number of components in this ColorSpace.

Overrides:

fromCIEXYZ in class ColorSpace

Class java.awt.color.ICC_Profile

```
java.lang.Object
   |
   +----java.awt.color.ICC_Profile
```

public class **ICC_Profile**
extends Object
super class of:
    ICC_ProfileGray, ICC_ProfileRGB

A representation of color profile data for device independent and device dependent color spaces based on the ICC Profile Format Specification, Version 3.3, November 11, 1996, from the International Color Consortium (see http://www.color.org).

An ICC_ColorSpace object can be constructed from an appropriate ICC_Profile. Typically, an ICC_ColorSpace would be associated with an ICC Profile which is either an input, display, or output profile (see the ICC specification). There are also device link, abstract, color space conversion, and named color profiles. These are less useful for tagging a color or image, but are useful for other purposes (in particular device link profiles can provide improved performance for converting from one device's color space to another's).

ICC Profiles represent transformations from the color space of the profile (e.g. a monitor) to a Profile Connection Space (PCS). Profiles of interest for tagging images or colors have a PCS which is one of the two specific device independent spaces (one CIEXYZ space and one CIELab space) defined in the ICC Profile Format Specification. Most profiles of interest either have invertible transformations or explicitly specify transformations going both directions.

See Also:
    ICC_ColorSpace

Variable Index

o **CLASS_ABSTRACT**
    Profile class is abstract.
o **CLASS_COLORSPACECONVERSION**
    Profile class is color space conversion.

o **CLASS_DEVICELINK**
Profile class is device link.
o **CLASS_DISPLAY**
Profile class is display.
o **CLASS_INPUT**
Profile class is input.
o **CLASS_NAMEDCOLOR**
Profile class is named color.
o **CLASS_OUTPUT**
Profile class is output.

Constructor Index

o **ICC_Profile**()
Constructs an empty ICC_Profile object.

Method Index

o **getColorSpaceType**()
Returns the color space type.
o **getData**(String)
Returns a particular tagged data element from the profile as a byte array.
o **getInstance**(byte[])
Returns an ICC_Profile corresponding to the data in a byte array.
o **getInstance**(InputStream)
Returns an ICC_Profile corresponding to the data in an InputStream.
o **getInstance**(int)
Returns an ICC_Profile corresponding to one of the specific color spaces defined by the ColorSpace class (for example CS_sRGB).
o **getMajorVersion**()
Returns profile major version.
o **getMinorVersion**()
Returns profile minor version.
o **getNumComponents**()
Returns the number of color components in the "input" color space of this profile.
o **getPCSType**()
Returns the color space type of the Profile Connection Space (PCS).
o **getProfileClass**()
Returns the profile class.
o **setData**(String, byte[])
Sets a particular tagged data element in the profile from a byte array.

Variables

o **CLASS_INPUT**

```
public static final int CLASS_INPUT
```

Profile class is input.

o **CLASS_DISPLAY**

```
public static final int CLASS_DISPLAY
```

Profile class is display.

o **CLASS_OUTPUT**

```
public static final int CLASS_OUTPUT
```

Profile class is output.

o **CLASS_DEVICELINK**

```
public static final int CLASS_DEVICELINK
```

Profile class is device link.

o **CLASS_COLORSPACECONVERSION**

```
public static final int CLASS_COLORSPACECONVERSION
```

Profile class is color space conversion.

o **CLASS_ABSTRACT**

```
public static final int CLASS_ABSTRACT
```

Profile class is abstract.

o **CLASS_NAMEDCOLOR**

```
public static final int CLASS_NAMEDCOLOR
```

Profile class is named color.

Constructors

o **ICC_Profile**

```
public ICC_Profile()
```

Constructs an empty ICC_Profile object.

Methods

o **getInstance**

```
public static ICC_Profile getInstance(int cspace)
```

Returns an ICC_Profile corresponding to one of the specific color spaces defined by the ColorSpace class (for example CS_sRGB).

o **getInstance**

```
public static ICC_Profile getInstance(byte data[])
```

Returns an ICC_Profile corresponding to the data in a byte array.

o **getInstance**

```
public static ICC_Profile getInstance(InputStream s)
```

Returns an ICC_Profile corresponding to the data in an InputStream.

o **getMajorVersion**

```
public int getMajorVersion()
```

Returns profile major version.

o **getMinorVersion**

```
public int getMinorVersion()
```

Returns profile minor version.

o **getProfileClass**

```
public int getProfileClass()
```

Returns the profile class. Returns one of the predefined profile class constants.

o **getColorSpaceType**

```
public int getColorSpaceType()
```

Returns the color space type. Returns one of the color space type constants defined by the ColorSpace class. This is the "input" color space of the profile. The type defines the number of components of the color space and the interpretation, e.g. TYPE_RGB identifies a color space with three components – red, green, and blue. It does not define the particular color characteristics of the space, e.g. the chromaticities of the primaries.

o **getPCSType**

```
public int getPCSType()
```

Returns the color space type of the Profile Connection Space (PCS). Returns one of the color space type constants defined by the ColorSpace class. This is the "output" color space of the profile. For an input, display, or output profile useful for tagging colors or images, this will be either TYPE_XYZ or TYPE_Lab and should be interpreted as the corresponding specific color space defined in the ICC specification. For a device link profile, this could be any of the color space type constants.

o **getData**

```
public byte[] getData(String s)
```

Returns a particular tagged data element from the profile as a byte array. Elements are identified by strings of four letters as defined in the ICC specification. The string "head" can be used to get the header. This method is useful for advanced applets or applications which need to access profile data directly. Returns null if no such tag exists.

o **setData**

```
public void setData(String s,
                    byte iccdata[])
```

Sets a particular tagged data element in the profile from a byte array. Elements are identified by strings of four letters as defined in the ICC specification. The string "head" can be used to set the header. This method is useful for advanced applets or applications which need to access profile data directly.

o **getNumComponents**

```
public int getNumComponents()
```

Returns the number of color components in the "input" color space of this profile. For example if the color space type of this profile is TYPE_RGB, then this method will return 3.

Class java.awt.color.ICC_ProfileGray

```
java.lang.Object
   |
   +----java.awt.color.ICC_Profile
           |
           +----java.awt.color.ICC_ProfileGray
```

public class **ICC_ProfileGray**
extends ICC_Profile

A subclass of the ICC_Profile class which represents profiles which meet the following criteria: the color space type of the profile is TYPE_GRAY, the PCS is the CIEXYZ space defined by the ICC specification, and the profile includes the grayTRCTag and mediaWhitePointTag tags. Examples of this kind of profile are monochrome input profiles, monochrome display profiles, and monochrome output profiles. The ICC_Profile.getInstance() method will return an ICC_ProfileGray object when the above conditions are met. The advantage of this class is that it provides a lookup table that Java or native methods may be able to use directly to optimize color conversion in some cases.

To transform from a GRAY device profile color space to the CIEXYZ Profile Connection Space, the device gray component is transformed by a lookup through the tone reproduction curve (TRC). The result is treated as the achromatic component of the PCS.

```
PCSY = grayTRC[deviceGray]
```

The inverse transform is done by converting the PCS Y components to device Gray via the inverse of the grayTRC.

Constructor Index

o **ICC_ProfileGray**()
    Constructs an empty ICC_ProfileGray.
o **ICC_ProfileGray**(byte[])
    Constructs a new ICC_ProfileGray from a byte data array.

Method Index

o **getGamma**()
: Returns a gamma value representing the tone reproduction curve (TRC).

o **getMediaWhitePoint**()
: Returns a float array of length 3 containing the X, Y, and Z components of the mediaWhitePointTag in the ICC profile.

o **getTRC**()
: Returns the TRC as an array of shorts.

o **getTRCLUT**(boolean, int, int)
: Returns a LookupTable object which represents the TRC.

Constructors

o **ICC_ProfileGray**

```
public ICC_ProfileGray()
```

Constructs an empty ICC_ProfileGray. Before using the profile for any purpose all necessary profile data must be set to appropriate values.

o **ICC_ProfileGray**

```
public ICC_ProfileGray(byte data[])
```

Constructs a new ICC_ProfileGray from a byte data array.

Methods

o **getMediaWhitePoint**

```
public float[] getMediaWhitePoint()
```

Returns a float array of length 3 containing the X, Y, and Z components of the mediaWhitePointTag in the ICC profile.

o **getGamma**

```
public float getGamma()
```

Returns a gamma value representing the tone reproduction curve (TRC). If the profile represents the TRC as a table rather than a single gamma value, then an exception is thrown. In this case the actual table can be obtained via getTRC(). When using a gamma value, the PCS Y component is computed as follows:

```
gamma
PCSY = deviceGray
```

Returns:

the gamma value as a float.

Throws: NoSuchProfileDataException

if the profile does not specify the TRC as a single gamma value.

o **getTRC**

```
public short[] getTRC()
```

Returns the TRC as an array of shorts. If the profile has specified the TRC as linear (gamma = 1.0) or as a simple gamma value, this method throws an exception, and the getGamma() method should be used to get the gamma value. Otherwise the short array returned here represents a lookup table where the input Gray value is conceptually in the range [0.0, 1.0]. Value 0.0 maps to array index 0 and value 1.0 maps to array index length−1. Interpolation may be used to generate output values for input values which do not map exactly to an index in the array. Output values also map linearly to the range [0.0, 1.0]. Value 0.0 is represented by an array value of 0x0000 and value 1.0 by 0xFFFF, i.e. the values are really unsigned short values, although they are returned in a short array.

Returns:

a short array representing the TRC.

Throws: NoSuchProfileDataException

if the profile does not specify the TRC as a table.

o **getTRCLUT**

```
public LookupTable getTRCLUT(boolean shortdata,
                             int num_entries,
                             int offset)
```

Returns a LookupTable object which represents the TRC. If shortdata is true, a ShortLookupTable is returned. Otherwise, a ByteLookupTable is returned. The number of entries and offset for the LookupTable are provided as arguments to this method. The first entry in the LookupTable will correspond to a TRC input value of 0.0, and the last entry to 1.0. The byte or short entries in the LookupTable will be computed from the TRC of the profile and are unsigned values ranging from 0x00 to 0xFF for byte data and from 0x0000 to 0xFFFF for short data. This method returns a valid LookupTable both for TRCs specified as tables and those specified as gamma values in the profile.

Returns:

a LookupTable representing the TRC.

Class java.awt.color.ICC_ProfileRGB

```
java.lang.Object
   |
   +----java.awt.color.ICC_Profile
           |
           +----java.awt.color.ICC_ProfileRGB
```

---

public class **ICC_ProfileRGB**
extends ICC_Profile

A subclass of the ICC_Profile class which represents profiles which meet the following criteria: the color space type of the profile is RGB, the PCS is the CIEXYZ space defined by the ICC specification, and the profile includes the redColorantTag, greenColorantTag, blueColorantTag, redTRCTag, greenTRCTag, blueTRCTag, and mediaWhitePointTag tags. Examples of this kind of profile are three−component matrix−based input profiles and RGB display profiles. The ICC_Profile.getInstance() method will return an ICC_ProfileRGB object when the above conditions are met. The advantage of this class is that it provides color transform matrices and lookup tables that Java or native methods may be able to use directly to optimize color conversion in some cases.

To transform from a device profile color space to the CIEXYZ Profile Connection Space, each device color component is first linearized by a lookup through the corresponding tone reproduction curve (TRC). Then the resulting linear RGB components are converted via a 3x3 matrix (constructed from the RGB colorants) to the CIEXYZ PCS.

```
linearR = redTRC[deviceR]
linearG = greenTRC[deviceG]
linearB = blueTRC[deviceB]
_      _       _                                            _   _         _
[  PCSX  ]      [  redColorantX  greenColorantX  blueColorantX  ] [  linearR  ]
[        ]      [                                               ] [           ]
[  PCSY  ]  =  [  redColorantY  greenColorantY  blueColorantY  ] [  linearG  ]
[        ]      [                                               ] [           ]
[_ PCSZ _]      [_ redColorantZ  greenColorantZ  blueColorantZ _] [_ linearB _]
```

The inverse transform is done by converting PCS XYZ components to linear RGB components via the inverse of the above 3x3 matrix, and then converting linear RGB to device RGB via inverses of the TRCs.

Variable Index

o **ALLCOMPONENTS**
    To request a TRCLUT for the components.
o **BLUECOMPONENT**
    To request a gamma value, TRC, or TRCLUT for the blue component.
o **GREENCOMPONENT**
    To request a gamma value, TRC, or TRCLUT for the green component.
o **REDCOMPONENT**
    To request a gamma value, TRC, or TRCLUT for the red component.

Constructor Index

o **ICC_ProfileRGB**()
    Constructs an empty ICC_ProfileRGB.
o **ICC_ProfileRGB**(byte[])
    Constructs a new ICC_ProfileRGB from a byte data array.

Method Index

o **getGamma**(int)
    Returns a gamma value representing the tone reproduction curve (TRC) for a particular component.
o **getMatrix**()
    Returns a 3x3 float matrix constructed from the X, Y, and Z components of the redColorantTag, greenColorantTag, and blueColorantTag in the ICC profile, as described above.
o **getMediaWhitePoint**()
    Returns a float array of length 3 containing the X, Y, and Z components of the mediaWhitePointTag in the ICC profile.
o **getTRC**(int)
    Returns the TRC for a particular component as an array of shorts.
o **getTRCLUT**(int, boolean, int, int)
    Returns a LookupTable object which represents the TRC for a single component (specify REDCOMPONENT, GREENCOMPONENT, or BLUECOMPONENT) or for all three components (specify ALLCOMPONENTS).

Variables

o **REDCOMPONENT**

```
public static final int REDCOMPONENT
```

To request a gamma value, TRC, or TRCLUT for the red component.

o **GREENCOMPONENT**

```
public static final int GREENCOMPONENT
```

To request a gamma value, TRC, or TRCLUT for the green component.

o **BLUECOMPONENT**

```
public static final int BLUECOMPONENT
```

To request a gamma value, TRC, or TRCLUT for the blue component.

o **ALLCOMPONENTS**

```
public static final int ALLCOMPONENTS
```

To request a TRCLUT for the components.

Constructors

o **ICC_ProfileRGB**

```
public ICC_ProfileRGB()
```

Constructs an empty ICC_ProfileRGB. Before using the profile for any purpose all necessary profile data must be set to appropriate values.

o **ICC_ProfileRGB**

```
public ICC_ProfileRGB(byte data[])
```

Constructs a new ICC_ProfileRGB from a byte data array.

Methods

o **getMediaWhitePoint**

```
public float[] getMediaWhitePoint()
```

Returns a float array of length 3 containing the X, Y, and Z components of the mediaWhitePointTag in the ICC profile.

o **getMatrix**

```
public float[][] getMatrix()
```

Returns a 3x3 float matrix constructed from the X, Y, and Z components of the redColorantTag, greenColorantTag, and blueColorantTag in the ICC profile, as described above. This matrix can be used for color transforms in the forward direction of the profile, i.e.

from the profile color space to the CIEXYZ PCS.

o **getGamma**

```
public float getGamma(int component)
```

Returns a gamma value representing the tone reproduction curve (TRC) for a particular component. Component must be one of REDCOMPONENT, GREENCOMPONENT, or BLUECOMPONENT. If the profile represents the TRC for the corresponding component as a table rather than a single gamma value, then an exception is thrown. In this case the actual table can be obtained via getTRC(). When using a gamma value, the linear component (R, G, or B) is computed as follows:

```
gamma
linearComponent = deviceComponent
```

Returns:
the gamma value as a float.
Throws: NoSuchProfileDataException
if the profile does not specify the corresponding TRC as a single gamma value.

o **getTRC**

```
public short[] getTRC(int component)
```

Returns the TRC for a particular component as an array of shorts. Component must be one of REDCOMPONENT, GREENCOMPONENT, or BLUECOMPONENT. If the profile has specified the corresponding TRC as linear (gamma = 1.0) or as a simple gamma value, this method throws an exception, and the getGamma() method should be used to get the gamma value. Otherwise the short array returned here represents a lookup table where the input component value is conceptually in the range [0.0, 1.0]. Value 0.0 maps to array index 0 and value 1.0 maps to array index length−1. Interpolation may be used to generate output values for input values which do not map exactly to an index in the array. Output values also map linearly to the range [0.0, 1.0]. Value 0.0 is represented by an array value of 0x0000 and value 1.0 by 0xFFFF, i.e. the values are really unsigned short values, although they are returned in a short array.

Returns:
a short array representing the TRC.
Throws: NoSuchProfileDataException
if the profile does not specify the corresponding TRC as a table.

o **getTRCLUT**

```
public LookupTable getTRCLUT(int component,
                             boolean shortdata,
                             int num_entries,
                             int offset)
```

Returns a LookupTable object which represents the TRC for a single component (specify REDCOMPONENT, GREENCOMPONENT, or BLUECOMPONENT) or for all three components (specify ALLCOMPONENTS). If shortdata is true, a ShortLookupTable is returned. Otherwise, a ByteLookupTable is returned. The number of entries and offset for the LookupTable are provided as arguments to this method. The first entry in the LookupTable will correspond to a TRC input value of 0.0, and the last entry to 1.0. The byte or short entries in the LookupTable will be computed from the TRCs of the profile and are unsigned values ranging from 0x00 to 0xFF for byte data and from 0x0000 to 0xFFFF for short data. This method returns a valid LookupTable both for TRCs specified as tables and those specified as gamma values in the profile.

Returns:

a LookupTable representing the TRC(s).

Class java.awt.color.NoSuchProfileDataException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----java.lang.RuntimeException
                           |
                           +----java.awt.color.NoSuchProfileDataException
```

public class **NoSuchProfileDataException**
extends RuntimeException

This exception is thrown if an attempt is made to access data in an ICC_Profile object when such data doesn't exist.

Constructor Index

o **NoSuchProfileDataException**(String)

Constructors

o **NoSuchProfileDataException**

```
 public NoSuchProfileDataException(String s)
```

Interface java.awt.font.MultipleMaster

public interface **MultipleMaster**

This is an interface representing Type 1 Multiple Master fonts. A particular Font object may implement this interface.

Method Index

o **deriveMMFont**(float[])
  Creates a new instance of a multiple master font based on the design axes values specified by the array.
o **deriveMMFont**(float[], float, float, float, float)
  Creates a new instance of a multiple master font based on detailed metric information.
o **getDesignAxisDefaults**()
  Returns an array of default design values for each axis.
o **getDesignAxisNames**()
  Returns the name for each design axis.
o **getDesignAxisRanges**()
  Returns an array of design limits interleaved in the form [from−>to] for each axis.
o **getNumDesignAxes**()
  Returns the number of multiple master design controls.

Methods

o **getNumDesignAxes**

```
public abstract int getNumDesignAxes()
```

Returns the number of multiple master design controls. Design axes include things like width, weight and optical scaling.

o **getDesignAxisRanges**

```
public abstract float[] getDesignAxisRanges()
```

Returns an array of design limits interleaved in the form [from−>to] for each axis. For example, design limits for weight could be from 0.1 to 1.0. The values will be returned in the same order returned by getDesignAxisNames().

o **getDesignAxisDefaults**

```
public abstract float[] getDesignAxisDefaults()
```

Returns an array of default design values for each axis. For example, the default value for weight could be 1.6. The values will be returned in the same order returned by getDesignAxisNames().

o **getDesignAxisNames**

```
public abstract String[] getDesignAxisNames()
```

Returns the name for each design axis. This also determines the order in which the values for each axis will be returned.

o **deriveMMFont**

```
public abstract Font deriveMMFont(float axes[])
```

Creates a new instance of a multiple master font based on the design axes values specified by the array. The size of the array must correspond to the value returned from getNumDesignAxes() and the values of the array elements must fall within limits specified by getDesignAxesLimits(). In case of an error, NULL is returned.

o **deriveMMFont**

```
public abstract Font deriveMMFont(float glyphWidths[],
                                  float avgStemWidth,
                                  float typicalCapHeight,
                                  float typicalXHeight,
                                  float italicAngle)
```

Creates a new instance of a multiple master font based on detailed metric information. In case of an error, NULL is returned.

Parameters:

glyphWidths – An array of floats representing the desired width of each glyph in font space.
avgStemWidth – The average stem width for the overall font in font space.
typicalCapHeight – The height of a typical upper case char.
typicalXHeight – The height of a typical lower case char.
italicAngle – The angle at which the italics lean, in degrees counterclockwise from vertical.

All Packages Class Hierarchy This Package Previous Next Index

---

Interface java.awt.font.OpenType

public interface **OpenType**

This interface represents OpenType and TrueType fonts. The goal for this interface is to be able to return *sfnt* tables from the font. A particular Font object may implement this interface.

Variable Index

o **TAG_ACNT**
  Accent attachment.
o **TAG_AVAR**
  Axis variaiton.
o **TAG_BASE**
  Baseline data.
o **TAG_BDAT**
  Bitmap data.
o **TAG_BLOC**
  Bitmap location.
o **TAG_BSLN**
  Baselint table.
o **TAG_CFF**
  Compact font format (Type1 font).
o **TAG_CMAP**
  Character to glyph mapping.
o **TAG_CVAR**
  CVT variation.
o **TAG_CVT**
  Control value table.
o **TAG_DSIG**
  Digital signature.
o **TAG_EBDT**
  Embedded bitmap data.
o **TAG_EBLC**
  Embedded bitmap location.
o **TAG_EBSC**
  Embedded bitmap scaling.
o **TAG_FDSC**
  Font descriptors.
o **TAG_FEAT**

Feature name.

o **TAG_FMTX**
Font metrics.

o **TAG_FPGM**
Font program.

o **TAG_FVAR**
Font variation.

o **TAG_GASP**
Grid_fitting and scan conversion procedure.

o **TAG_GDEF**
Glyph definition.

o **TAG_GLYF**
Glyph data.

o **TAG_GPOS**
Glyph positioning.

o **TAG_GSUB**
Glyph substitution.

o **TAG_GVAR**
Glyph variation.

o **TAG_HDMX**
Horizontal device metrics.

o **TAG_HEAD**
Font header.

o **TAG_HHEA**
Horizontal metrics header.

o **TAG_HMTX**
Horizontal metrics.

o **TAG_JSTF**
Justification.

o **TAG_JUST**
Justification.

o **TAG_KERN**
Kerning.

o **TAG_LCAR**
Ligature caret.

o **TAG_LOCA**
Index to location.

o **TAG_LTSH**
Linear threshold.

o **TAG_MAXP**
Maximum profile.

o **TAG_MMFX**
Multiple master font metrics.

o **TAG_MMSD**
Multiple master supplementary data.

o **TAG_MORT**
Glyph metamorphosis.

o **TAG_NAME**
Naming table.

- o **TAG_OPBD**
  Optical bounds.
- o **TAG_OS2**
  OS/2 and windows specific metrics.
- o **TAG_PCLT**
  PCL 5 data.
- o **TAG_POST**
  Postscript Information.
- o **TAG_PREP**
  CVT preprogram.
- o **TAG_PROP**
  Glyph properties.
- o **TAG_TRAK**
  Tracking.
- o **TAG_TYP1**
  Adobe Type 1 font data.
- o **TAG_VDMX**
  Vertical device metrics.
- o **TAG_VHEA**
  Vertical metrics header.
- o **TAG_VMTX**
  Vertical metrics.

Method Index

- o **getFontTable**(int)
  Returns the table as an array of bytes for a given tag.
- o **getVersion**()
  Returns the version of the OpenType font.

Variables

o **TAG_CMAP**

```
public static final int TAG_CMAP
```

Character to glyph mapping.

o **TAG_HEAD**

```
public static final int TAG_HEAD
```

Font header.

o **TAG_NAME**

```
public static final int TAG_NAME
```

Naming table.

o **TAG_GLYF**

```
public static final int TAG_GLYF
```

Glyph data.

o **TAG_MAXP**

```
public static final int TAG_MAXP
```

Maximum profile.

o **TAG_PREP**

```
public static final int TAG_PREP
```

CVT preprogram.

o **TAG_HMTX**

```
public static final int TAG_HMTX
```

Horizontal metrics.

o **TAG_KERN**

```
public static final int TAG_KERN
```

Kerning.

o **TAG_HDMX**

```
public static final int TAG_HDMX
```

Horizontal device metrics.

o **TAG_LOCA**

```
public static final int TAG_LOCA
```

Index to location.

o **TAG_POST**

```
public static final int TAG_POST
```

Postscript Information.

o **TAG_OS2**

```
public static final int TAG_OS2
```

OS/2 and windows specific metrics.

o **TAG_CVT**

```
public static final int TAG_CVT
```

Control value table.

o **TAG_GASP**

```
public static final int TAG_GASP
```

Grid_fitting and scan conversion procedure.

o **TAG_VDMX**

```
public static final int TAG_VDMX
```

Vertical device metrics.

o **TAG_VMTX**

```
public static final int TAG_VMTX
```

Vertical metrics.

o **TAG_VHEA**

```
public static final int TAG_VHEA
```

Vertical metrics header.

o **TAG_HHEA**

```
public static final int TAG_HHEA
```

Horizontal metrics header.

o **TAG_TYP1**

```
public static final int TAG_TYP1
```

Adobe Type 1 font data.

o **TAG_BSLN**

```
public static final int TAG_BSLN
```

Baselint table.

o **TAG_GSUB**

```
public static final int TAG_GSUB
```

Glyph substitution.

o **TAG_DSIG**

```
public static final int TAG_DSIG
```

Digital signature.

o **TAG_FPGM**

```
public static final int TAG_FPGM
```

Font program.

o **TAG_FVAR**

```
public static final int TAG_FVAR
```

Font variation.

o **TAG_GVAR**

```
public static final int TAG_GVAR
```

Glyph variation.

o **TAG_CFF**

```
public static final int TAG_CFF
```

Compact font format (Type1 font).

o **TAG_MMSD**

```
public static final int TAG_MMSD
```

Multiple master supplementary data.

o **TAG_MMFX**

```
public static final int TAG_MMFX
```

Multiple master font metrics.

o **TAG_BASE**

```
public static final int TAG_BASE
```

Baseline data.

o **TAG_GDEF**

```
public static final int TAG_GDEF
```

Glyph definition.

o **TAG_GPOS**

```
public static final int TAG_GPOS
```

Glyph positioning.

o **TAG_JSTF**

```
public static final int TAG_JSTF
```

Justification.

o **TAG_EBDT**

```
public static final int TAG_EBDT
```

Embedded bitmap data.

o **TAG_EBLC**

```
public static final int TAG_EBLC
```

Embedded bitmap location.

o **TAG_EBSC**

```
public static final int TAG_EBSC
```

Embedded bitmap scaling.

o **TAG_LTSH**

```
public static final int TAG_LTSH
```

Linear threshold.

o **TAG_PCLT**

```
public static final int TAG_PCLT
```

PCL 5 data.

o **TAG_ACNT**

```
public static final int TAG_ACNT
```

Accent attachment.

o **TAG_AVAR**

```
public static final int TAG_AVAR
```

Axis variaiton.

o **TAG_BDAT**

```
public static final int TAG_BDAT
```

Bitmap data.

o **TAG_BLOC**

```
public static final int TAG_BLOC
```

Bitmap location.

o **TAG_CVAR**

```
public static final int TAG_CVAR
```

CVT variation.

o **TAG_FEAT**

```
public static final int TAG_FEAT
```

Feature name.

o **TAG_FDSC**

```
public static final int TAG_FDSC
```

Font descriptors.

o **TAG_FMTX**

```
public static final int TAG_FMTX
```

Font metrics.

o **TAG_JUST**

```
public static final int TAG_JUST
```

Justification.

o **TAG_LCAR**

```
public static final int TAG_LCAR
```

Ligature caret.

o **TAG_MORT**

```
public static final int TAG_MORT
```

Glyph metamorphosis.

o **TAG_OPBD**

```
public static final int TAG_OPBD
```

Optical bounds.

o **TAG_PROP**

```
public static final int TAG_PROP
```

Glyph properties.

o **TAG_TRAK**

```
public static final int TAG_TRAK
```

Tracking.

Methods

o **getVersion**

```
public abstract int getVersion()
```

Returns the version of the OpenType font. 1.0 is represented as 0x00010000.

o **getFontTable**

```
public abstract byte[] getFontTable(int sfntTag)
```

Returns the table as an array of bytes for a given tag. Tags for sfnt tables include items like *cmap*, *name* and *head*. The byte array returned is a copy of the font data in memory.

Class java.awt.font.GlyphMetrics

```
java.lang.Object
   |
   +----java.awt.font.GlyphMetrics
```

---

public final class **GlyphMetrics**
extends Object

This class provides metrics for a single glyph. This information defines how to place the glyphs. It does not take into account any kerning information.

See Also:
getGlyphMetrics

Variable Index

o **COMBINING**
This type indicates a glyph that represents a combining character, such as an umlaut.
o **COMPONENT**
This type indicates a glyph that is used in addition to the logically preceeding DEFAULT or LIGATURE glyph.
o **HANGING_PUNCTUATION**
This value is added to the base type to indicate hanging punctuation.
o **LIGATURE**
This type indicates a glyph that represents multiple characters as a ligature, i.e.
o **STANDARD**
This type indicates a glyph that represents a single standard spacing character.
o **WHITESPACE**
This value is added to the base type to indicate whitespace.

Constructor Index

o **GlyphMetrics**(float, Rectangle2D, byte)
Creates a GlyphMetrics object from an advance and bounds of a glyph.

Method Index

o **getAdvance**()
    Returns the device independent advance.
o **getBounds2D**()
    Returns the device independent bounding box.
o **getLSB**()
    Returns the left side bearing of the glyph.
o **getRSB**()
    Returns the right side bearing of the glyph.
o **getType**()
    Return the raw glyph type code.
o **isCombining**()
    Return true if this is a combining glyph.
o **isComponent**()
    Return true if this is a component glyph.
o **isHangingPunctuation**()
    Return true if this is a hanging punctuation glyph.
o **isLigature**()
    Return true if this is a ligature glyph.
o **isStandard**()
    Return true if this is a standard glyph.
o **isVertical**()
    Returns true if the metrics associated with this glyph are vertical and false if they are horizontal.
o **isWhitespace**()
    Return true if this is a whitespace glyph.

Variables

o **STANDARD**

```
public static final byte STANDARD
```

This type indicates a glyph that represents a single standard spacing character.

o **LIGATURE**

```
public static final byte LIGATURE
```

This type indicates a glyph that represents multiple characters as a ligature, i.e. 'fi' or 'ffi' It is followed by non−spacing, non−drawing combining glyphs for the remaining characters.

o **COMBINING**

```
public static final byte COMBINING
```

This type indicates a glyph that represents a combining character, such as an umlaut.

o **COMPONENT**

```
public static final byte COMPONENT
```

This type indicates a glyph that is used in addition to the logically preceeding DEFAULT or LIGATURE glyph. It has no corresponding character in the backing store, but instead is associated with the characters represented by the base glyph.

o **HANGING_PUNCTUATION**

```
public static final byte HANGING_PUNCTUATION
```

This value is added to the base type to indicate hanging punctuation. This information is used in justification.

o **WHITESPACE**

```
public static final byte WHITESPACE
```

This value is added to the base type to indicate whitespace. This information is used in justification.

Constructors

o **GlyphMetrics**

```
public GlyphMetrics(float advance,
                    Rectangle2D bounds,
                    byte glyphType)
```

Creates a GlyphMetrics object from an advance and bounds of a glyph.

Parameters:

float – the advance width of the glyph.
bounds – the device independent bounding box of the glyph.
isVertical – it is true if the associated metrics are vertical.

Methods

o **getAdvance**

```
public float getAdvance()
```

Returns the device independent advance. This method takes point size into account. The advance returned is effectively in user space coordinates.

Returns:

the advance of the glyph

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Returns the device independent bounding box. This method takes point size into account. The bounding box returned is effectively in user space coordinates.

o **getLSB**

```
public float getLSB()
```

Returns the left side bearing of the glyph. This value is calculated.

o **getRSB**

```
public float getRSB()
```

Returns the right side bearing of the glyph. This value is calculated.

o **isVertical**

```
public boolean isVertical()
```

Returns true if the metrics associated with this glyph are vertical and false if they are horizontal.

o **getType**

```
public byte getType()
```

Return the raw glyph type code.

o **isStandard**

```
public boolean isStandard()
```

Return true if this is a standard glyph.

o **isLigature**

```
public boolean isLigature()
```

Return true if this is a ligature glyph.

o **isCombining**

```
public boolean isCombining()
```

Return true if this is a combining glyph.

o **isComponent**

```
public boolean isComponent()
```

Return true if this is a component glyph.

o **isWhitespace**

```
public boolean isWhitespace()
```

Return true if this is a whitespace glyph.

o **isHangingPunctuation**

```
public boolean isHangingPunctuation()
```

Return true if this is a hanging punctuation glyph.

Class java.awt.font.GlyphSet

```
java.lang.Object
   |
   +----java.awt.font.GlyphSet
```

---

public class **GlyphSet**
extends Object

This is the lowest level representation used to draw text. Exact glyph codes and positions are used.

See Also:
drawString

Variable Index

o **LEFT_CONNECT_MASK**
Flag indicating a glyph is graphically connected to the preceeding glyph.
o **LEFT_SHAPED_MASK**
Flag indicating a glyph's shape has been modifed by the glyph preceeding it.
o **NO_MASK**
Flag indicating a glyph is graphically unaffected by surrounding glyphs.
o **RIGHT_CONNECT_MASK**
Flag indicating a glyph is graphically connected to the following glyph.
o **RIGHT_SHAPED_MASK**
Flag indicating a glyph's shape has been modifed by the glyph following it.

Constructor Index

o **GlyphSet**(Font, int[], byte, float, float[], float, float[], int[], byte[], int)
Constructs a new GlyphSet.

Method Index

o **appendGlyphs**(int, int[], float[], float[], byte[], int[])
Append glyphs to this GlyphSet.

- o **applyJustification**(float[], int, boolean[])
  Return a justified copy of this glyphset.
- o **clone**()
- o **equals**(GlyphSet)
- o **equals**(Object)
- o **getAdvance**()
  Get the advance of the GlyphSet along its baseline.
- o **getAdvanceBetween**(int, int)
  Return the sum of the advances for the glyphs corresponding to characters in logical positions from start up to limit.
- o **getAdvances**()
- o **getAdvances**(boolean)
- o **getAscent**()
  Get the maximum ascent of the glyphs in a GlyphSet.
- o **getBackMapping**()
  Returns the backmapping for the GlyphSet.
- o **getBaseline**()
  Return the baseline used by this glyphset.
- o **getDescent**()
  Get the maximum descent of the glyphs in a GlyphSet.
- o **getFont**()
  Returns the Font associated with this GlyphSet.
- o **getGlyphCodes**()
  Returns an int array of the glyph codes for this GlyphSet.
- o **getInverseOrder**(int[])
  Return the inverse array, source array must map 1−1 i.e.
- o **getLevels**()
  Return an array indicating the direction levels of each glyph.
- o **getLineBreakIndex**(int, float)
- o **getNormalizedOrder**(int[], int, int)
  Return an array containing the values from start up to limit, normalized to fall within the range from 0 up to limit − start.
- o **getNumCharacters**()
  Return the number of characters represented by this glyphset.
- o **getNumGlyphs**()
  Returns the number of glyphs associated with this GlyphSet.
- o **getVisualOrder**()
  Return a mapping array from visual order to logical order.
- o **getXAdjust**()
- o **getXAdvances**(boolean)
  Returns the advance of each glyph along the X axis.
- o **getYAdjust**()
  Return the adjustment to the y position of the glyphset.
- o **getYAdvances**(boolean)
  Returns the advance of each glyph along the Y axis.
- o **hashCode**()
- o **isVertical**()
- o **subset**(int, int)
- o **toString**()

Variables

o **NO_MASK**

```
public static final int NO_MASK
```

Flag indicating a glyph is graphically unaffected by surrounding glyphs.

o **RIGHT_CONNECT_MASK**

```
public static final int RIGHT_CONNECT_MASK
```

Flag indicating a glyph is graphically connected to the following glyph.

o **LEFT_CONNECT_MASK**

```
public static final int LEFT_CONNECT_MASK
```

Flag indicating a glyph is graphically connected to the preceeding glyph.

o **RIGHT_SHAPED_MASK**

```
public static final int RIGHT_SHAPED_MASK
```

Flag indicating a glyph's shape has been modifed by the glyph following it.

o **LEFT_SHAPED_MASK**

```
public static final int LEFT_SHAPED_MASK
```

Flag indicating a glyph's shape has been modifed by the glyph preceeding it.

Constructors

o **GlyphSet**

```
public GlyphSet(Font font,
                int glyphCodes[],
                byte baseline,
                float xAdjust,
                float xAdvances[],
                float yAdjust,
                float yAdvances[],
                int visualOrder[],
                byte levels[],
                int glyphCount)
```

Constructs a new GlyphSet.

Parameters:

font – the Font object for the GlyphSet from which the glyphs are taken.
size – the mininum size of array parameters. Elements in the input arrays at index size and beyond are ignored by the GlyphSet.
glyphCodes – an int array of glyph codes representing specific glyphs in the font. The array must be non–null.
advanceWidths – the advances of the glyphs along the x–axis. All values must be non–negative. If null then default values will be generated.
flags – an array of flags indicating how each glyph's shape is influenced by its neighboring glyphs. Null may be passed if all values are set to NOT_CONNECTED_MASK.
backMapping – an int array representing the mapping from logical to visual order. If null, each glyph represents exactly one character in the text and the visual and logical order are assumed to be the same.

Methods

o **getInverseOrder**

```
public static int[] getInverseOrder(int values[])
```

Return the inverse array, source array must map 1–1 i.e. if values[i] = j, then inverse[j] = i.

o **getNormalizedOrder**

```
public static int[] getNormalizedOrder(int values[],
                                       int start,
                                       int limit)
```

Return an array containing the values from start up to limit, normalized to fall within the range from 0 up to limit – start. If this would be a canonical ltr ordering, return null.

o **getFont**

```
public Font getFont()
```

Returns the Font associated with this GlyphSet.

Returns:

font object associated with this glyph set.

o **appendGlyphs**

```
public void appendGlyphs(int size,
                         int glyphCodes[],
                         float xAdvance[],
                         float yAdvance[],
                         byte flags[],
                         int backMapping[])
```

Append glyphs to this GlyphSet.

Parameters:

size − the mininum size of array parameters. Elements in the input arrays at index size and beyond are ignored by the GlyphSet.
glyphCodes − an int array of glyph codes representing specific glyphs in the font. The array must be non−null.
xAdvance − the advance of the glyphs along the x−axis. All values must be non−negative. If null then default values will be generated.
yAdvance − the advance of the glyphs along the y−axis. All values must be non−negative. If null then default values will be generated.
flags − an array of flags indicating how each glyph's shape is influenced by its neighboring glyphs. Null may be passed if all values are set to NOT_CONNECTED_MASK.
backMapping − an int array representing the mapping from logical to visual order. If null, each glyph represents exactly one character in the text and the visual and logical order are assumed to be the same.

o **clone**

```
public Object clone()
```

Overrides:

clone in class Object

o **getAdvance**

```
public float getAdvance()
```

Get the advance of the GlyphSet along its baseline.

o **getAscent**

```
public float getAscent()
```

Get the maximum ascent of the glyphs in a GlyphSet.

o **getDescent**

```
public float getDescent()
```

Get the maximum descent of the glyphs in a GlyphSet.

o **getNumGlyphs**

```
public int getNumGlyphs()
```

Returns the number of glyphs associated with this GlyphSet.

o **getNumCharacters**

```
public int getNumCharacters()
```

Return the number of characters represented by this glyphset. Kashida justification may lead to there being more glyphs than characters. There are never fewer glyphs than characters.

o **getGlyphCodes**

```
public int[] getGlyphCodes()
```

Returns an int array of the glyph codes for this GlyphSet.

Returns:
an array of glyph codes associated with this GlyphSet.

o **getXAdvances**

```
public float[] getXAdvances(boolean mustBeNonNull)
```

Returns the advance of each glyph along the X axis. If non−null, only the first getNumGlyphs glyph codes are valid. If null, all glyphs advance 0 along the X axis. This array reflects the actual internal structure of the glyph set and any modification to it will be reflected in the glyph set itself.

o **getYAdvances**

```
public float[] getYAdvances(boolean mustBeNonNull)
```

Returns the advance of each glyph along the Y axis. If non−null, only the first getNumGlyphs glyph codes are valid. If null, all glyphs advance 0 along the Y axis. This array reflects the actual internal structure of the glyph set and any modification to it will be reflected in the glyph set itself.

o **getBaseline**

```
public byte getBaseline()
```

Return the baseline used by this glyphset. All glyphs in a single glyphset align to the same baseline.

See Also:
Font.getBaselineFor

o **getXAdjust**

```
public float getXAdjust()
```

o **getYAdjust**

```
public float getYAdjust()
```

Return the adjustment to the y position of the glyphset. The first visual glyph in the glyphset is positioned at offset by x, y from the origin of the glyphset. This supports such features as superscript and hanging punctuation.

See Also:
getXAdjust

o **getBackMapping**

```
public int[] getBackMapping()
```

Returns the backmapping for the GlyphSet. If null, each glyph represents exactly one character in the text and the text's visual order and logical order are assumed to be the same.

o **isVertical**

```
public boolean isVertical()
```

o **getAdvances**

```
public float[] getAdvances()
```

o **getAdvances**

```
public float[] getAdvances(boolean mustNotBeNull)
```

o **getVisualOrder**

```
public int[] getVisualOrder()
```

Return a mapping array from visual order to logical order. This array may be null, in which case the visual order == the logical order. Otherwise, the index used to access information about the leftmost (topmost) visual glyphs is provided by getVisualOrder()[0], the next glyph by getVisualOrder()[1], and so on.

The array may be longer than necessary. The actual number of valid values in the array is provided by getNumGlyphs().

See Also:
getNumGlyphs

o **getLevels**

```
public byte[] getLevels()
```

Return an array indicating the direction levels of each glyph. Glyphs whose level is even run from left to right (top to bottom), glyphs whose level is odd run from right to left (bottom to top). This array may be null, in which case the level of each glyph is zero.

The array may be longer than necessary. The actual number of valid values in the array is provided by getNumGlyphs().

See Also:
getNumGlyphs

o **applyJustification**

```
public GlyphSet applyJustification(float deltas[],
                                   int index,
                                   boolean flags[])
```

Return a justified copy of this glyphset.

See Also:
Font.handleApplyJustification

o **getAdvanceBetween**

```
public float getAdvanceBetween(int start,
                               int limit)
```

Return the sum of the advances for the glyphs corresponding to characters in logical positions from start up to limit. Note these are character indices, not glyph indices. They will be mapped internally to glyph indices.

Parameters:
start – the character index at which to start measuring
limit – the character index at which to stop measuring

o **getLineBreakIndex**

```
public int getLineBreakIndex(int start,
                             float hitAdvance)
```

o **subset**

```
public GlyphSet subset(int logicalStart,
                       int logicalLimit)
```

o **hashCode**

```
public int hashCode()
```

Overrides:
hashCode in class Object

o **equals**

```
public boolean equals(Object obj)
```

Overrides:
equals in class Object

o **equals**

```
public boolean equals(GlyphSet set)
```

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

Class java.awt.font.StyledString

```
java.lang.Object
   |
   +----java.awt.font.StyledString
```

---

public class **StyledString**
extends Object

This class encapsulates information needed to draw text. It simplifies drawing text strings with multiple Fonts and languages. It performs layout of glyphs representing text strings with multiple Fonts, including bi−directional layout for mixed languages.

See Also:
drawString

Constructor Index

- o **StyledString**()
  Constructs a new StyledString with no characters.
- o **StyledString**(String, Font)
  Constructs a new StyledString from the given String using the given Font.

Method Index

- o **charAt**(int)
  Returns the character at the specified index.
- o **concat**(StyledString)
  Concatenates the specified StyledString to the end of this StyledString and returns a new StyledString object representing the concatenation.
- o **createIterator**()
- o **createIterator**(int, int, int)
- o **getAdvanceVector**()
  Returns the advance vector for the end position of this StyledString.
- o **getAscent**()
  Returns the largest ascent of any font represented in this StyledString.

o **getAttributes**(int)
o **getBounds2D**()
    Returns the bounding box of this StyledString.
o **getDescent**()
    Returns the largest descent of any font represented in this StyledString.
o **getFontAt**(int)
    Returns the Font associated with the character at the specified index.
o **getGlyphSets**()
    Returns an array of GlyphSet objects associated with this StyledString.
o **getLeading**()
    Returns the largest leading of any font represented in this StyledString.
o **getRunLimit**(int)
o **getRunStart**(int)
o **getStringOutline**()
    Returns the Shape object for this StyledString, which is the union of all the outlines for glyphs in the StyledString.
o **length**()
    Returns the length of the StyledString.
o **substring**(int, int)
    Creates a StyledString object that is a substring of this StyledString.
o **toString**()
    Generates a String representation of the StyledString.

Constructors

o **StyledString**

```
public StyledString()
```

Constructs a new StyledString with no characters.

o **StyledString**

```
public StyledString(String str,
                    Font font)
```

Constructs a new StyledString from the given String using the given Font.

Parameters:
str – the String object.
font – the Font object.

Methods

o **substring**

```
public StyledString substring(int beginIndex,
                              int endIndex)
```

Creates a StyledString object that is a substring of this StyledString. The substring is specified by a beginIndex (inclusive) and an endIndex (exclusive).

Parameters:
beginIndex – the beginning character index (inclusive) for the substring.
endIndex – the ending character index (exclusive) for the substring.
Throws: StringIndexOutOfBoundsException
if the beginIndex or the endIndex is out of range, or if the beginIndex is greater than the endIndex.

o **getAscent**

```
public float getAscent()
```

Returns the largest ascent of any font represented in this StyledString. The units are user space coordinates.

o **getDescent**

```
public float getDescent()
```

Returns the largest descent of any font represented in this StyledString. The units are user space coordinates.

o **getLeading**

```
public float getLeading()
```

Returns the largest leading of any font represented in this StyledString. The units are user space coordinates.

o **getAttributes**

```
public AttributeSet getAttributes(int index)
```

o **getRunStart**

```
public int getRunStart(int index)
```

o **getRunLimit**

```
public int getRunLimit(int index)
```

o **getAdvanceVector**

```
public Point2D getAdvanceVector()
```

Returns the advance vector for the end position of this StyledString. The units are user space coordinates.

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Returns the bounding box of this StyledString. The units are user space coordinates.

o **getGlyphSets**

```
public GlyphSet[] getGlyphSets()
```

Returns an array of GlyphSet objects associated with this StyledString.

o **getStringOutline**

```
public Shape getStringOutline()
```

Returns the Shape object for this StyledString, which is the union of all the outlines for glyphs in the StyledString.

o **toString**

```
public String toString()
```

Generates a String representation of the StyledString.

Overrides:
toString in class Object

o **concat**

```
public StyledString concat(StyledString str)
```

Concatenates the specified StyledString to the end of this StyledString and returns a new StyledString object representing the concatenation.

o **charAt**

```
public char charAt(int index)
```

Returns the character at the specified index. An index ranges from 0 to length() – 1.

Throws: StringIndexOutOfBoundsException
if the index is not in the range 0 to length()–1.

o **getFontAt**

```
public Font getFontAt(int index)
```

Returns the Font associated with the character at the specified index. An index ranges from 0 to length() − 1.

Throws: StringIndexOutOfBoundsException
if the index is not in the range 0 to length()−1.

o **length**

```
public int length()
```

Returns the length of the StyledString. The length of the StyledString is equal to the number of 16−bit Unicode characters in the String.

o **createIterator**

```
public AttributedCharacterIterator createIterator()
```

o **createIterator**

```
public AttributedCharacterIterator createIterator(int start,
                                                  int limit,
                                                  int pos)
```

Class java.awt.font.TextAttributeSet

```
java.lang.Object
   |
   +----java.awt.font.TextAttributeSet
```

---

public final class **TextAttributeSet**
extends Object
implements MutableAttributeSet

This defines attributes and values used by the text layout classes. It also provides a simple implementation of MutableAttributeSet.

Variable Index

o **BACKGROUND**
  The background color.
o **BACKGROUND_DEFAULT**
  The default background color if BACKGROUND is unspecified.
o **BASELINE**
  Baseline to which to align lines.
o **BASELINE_DEFAULT**
  The default baseline if BASELINE is unspecified.
o **BASELINE_HANGING**
  Use a hanging baseline for the line.
o **BASELINE_IDEOGRAPHIC**
  Use a center baseline for the line.
o **BASELINE_OFFSETS**
  Baseline offsets used by a line.
o **BASELINE_OFFSETS_DEFAULT**
  The default baseline offsets if BASELINE_OFFSETS is unspecified.
o **BASELINE_ROMAN**
  Use a roman baseline for the line.
o **BIDI_EMBEDDING**
  The embedding level for nested bidirectional runs.
o **BIDI_EMBEDDING_DEFAULT**
  The default embedding if BIDI_EMBEDDING is unspecified.

o **BIDI_NUMERIC**
The bidirectional layout control for roman numeral text.
o **BIDI_NUMERIC_DEFAULT**
o **BIDI_NUMERIC_STRONG_ARABIC**
Interpret roman numerals as arabic numerals always.
o **BIDI_NUMERIC_STRONG_ROMAN**
Interpret roman numerals as roman numerals always.
o **BIDI_NUMERIC_WEAK_ARABIC**
Interpret roman numerals as having arabic line order.
o **EMPTY**
An empty attribute set, for convenience.
o **FAMILY**
The unlocalized font family name.
o **FAMILY_ALL**
o **FONT**
The font to use to render this text.
o **FOREGROUND**
The foreground color.
o **FOREGROUND_DEFAULT**
The default foreground color if FOREGROUND is unspecified.
o **INPUT_METHOD_SEGMENT**
Constant for the attribute "input method segment".
o **JUSTIFICATION**
Justification for the paragraph.
o **JUSTIFICATION_DEFAULT**
The default justification if JUSTIFICATION is unspecified.
o **JUSTIFICATION_FULL**
Justify the line to the full requested width.
o **LANGUAGE**
Constant for the attribute "language".
o **POSTURE**
The posture of the font.
o **POSTURE_DEFAULT**
The default posture if POSTURE is unspecified.
o **POSTURE_ITALIC**
The standard italic posture for the font.
o **READING**
Constant for the attribute "reading".
o **RUN_DIRECTION**
The run direction of the line, either left−to−right or right−to−left.
o **RUN_DIRECTION_DEFAULT**
The default direction if RUN_DIRECTION is unspecified.
o **RUN_DIRECTION_LTR**
Left−to−right (top−to−bottom) run direction.
o **RUN_DIRECTION_RTL**
Right−to−left (bottom−to−top) run direction.
o **SIZE**
The font size in points.
o **STRIKETHROUGH**

Adorn the text with strikethrough.

o **STRIKETHROUGH_DEFAULT**
    The default strikethrough if STRIKETHROUGH is unspecified, the value is false.

o **STRIKETHROUGH_ON**
    A single strikethrough.

o **SUPERSUBSCRIPT**
    Super and subscripting.

o **SUPERSUBSCRIPT_DEFAULT**
    The default supersubscript if SUPERSUBSCRIPT is unspecified.

o **SUPERSUBSCRIPT_DOWN1**
    Standard subscript.

o **SUPERSUBSCRIPT_UP1**
    Standard superscript.

o **TRANSFORM**
    The transform of the font.

o **TRANSFORM_DEFAULT**
    The default transform if TRANSFORM is unspecified.

o **UNDERLINE**
    Adorn the text with underlining.

o **UNDERLINE_DEFAULT**
    The default underline if UNDERLINE is unspecified.

o **UNDERLINE_ON**
    Draw an underline.

o **WEIGHT**
    The weight of the font.

o **WEIGHT_BOLD**
    The bold weight for the font.

o **WEIGHT_DEFAULT**
    The default weight if WEIGHT is unspecified.

o **WEIGHT_LIGHT**
    The light weight for the font.

Constructor Index

o **TextAttributeSet**()
    Construct a new mutable attribute set.

o **TextAttributeSet**(AttributeSet)
    Construct a new mutable attribute set containing the provided attributes.

o **TextAttributeSet**(String, Object)
    Construct a new mutable attribute set containing the provided attribute.

Method Index

o **add**(AttributeSet)
    Remove any existing attributes with the same names, and add the new attributes.

o **add**(String, Object)
    Remove any existing attribute with the same name, and add a new attribute with the given

name and value.

o **clone**()

Return a copy of this attribute set.

o **contains**(AttributeSet)

Returns true if this set contains all the attributes with equal values.

o **contains**(String, Object)

Returns true if this set contains this attribute with an equal value, or if the value is null and the attribute is not defined.

o **equals**(AttributeSet)

Return true if the argument contains the same attributes as this.

o **equals**(Object)

Return true if the argument is an attribute set containing the same attributes as this.

o **get**(String)

Returns the value of the attribute with this name, or null if the attribute is not defined.

o **getSize**()

Returns the number of attributes.

o **isEmpty**()

Returns true if the set is empty.

o **names**()

Returns an enumeration over the names of the attributes in the set.

o **remove**(Enumeration)

Remove any existing attributes with the given names.

o **remove**(String)

Remove any existing attribute with the given name.

o **set**(AttributeSet)

Removes all attributes, then adds the new attributes.

Variables

o **LANGUAGE**

```
public static final String LANGUAGE
```

Constant for the attribute "language". The value of this attribute should be an instance of Locale.

See Also:

Locale

o **READING**

```
public static final String READING
```

Constant for the attribute "reading". In languages where the written form and the pronunciation of a word are only loosely related (such as Japanese), it is often necessary to store the reading (pronunciation) along with the written form. This is an annotation attribute. The value should be an instance of Annotation holding an instance of String.

See Also:
Annotation, String

o **INPUT_METHOD_SEGMENT**

```
public static final String INPUT_METHOD_SEGMENT
```

Constant for the attribute "input method segment". Input methods often break up text into segments, which usually correspond to words. This is an annotation attribute. The value should be an instance of Annotation holding a value of null.

See Also:
String

o **FAMILY**

```
public static final String FAMILY
```

The unlocalized font family name.

Values are instances of String, e.g. "Serif", "Palatino".

o **FAMILY_ALL**

```
public static final String FAMILY_ALL
```

o **WEIGHT**

```
public static final String WEIGHT
```

The weight of the font.

Values are instances of Float. The value is between 0.0 and 2.0. This roughly corresponds to the ratio of the stem width to the width of regular characters for this typeface. Lower values are lighter.

o **WEIGHT_DEFAULT**

```
public static final Float WEIGHT_DEFAULT
```

The default weight if WEIGHT is unspecified. This value is 1.0.

See Also:
WEIGHT

o **WEIGHT_BOLD**

```
public static final Float WEIGHT_BOLD
```

The bold weight for the font. This value is 1.5.

See Also:
WEIGHT

o **WEIGHT_LIGHT**

```
public static final Float WEIGHT_LIGHT
```

The light weight for the font. This value is 0.7.

See Also:
WEIGHT

o **POSTURE**

```
public static final String POSTURE
```

The posture of the font.

Values are instances of Float. The value is the run/rise, generally between −1.0 and 1.0, where negative values indicate a leftwards slant, positive a rightwards slant, and zero no slant. There is no guarantee, however, that the slope of the text as returned by font will be the same as the posture value.

See Also:
getItalicAngle

o **POSTURE_DEFAULT**

```
public static final Float POSTURE_DEFAULT
```

The default posture if POSTURE is unspecified. This value is 0.0.

See Also:
POSTURE

o **POSTURE_ITALIC**

```
public static final Float POSTURE_ITALIC
```

The standard italic posture for the font. This value is 0.333.

See Also:
POSTURE

o **SIZE**

```
public static final String SIZE
```

The font size in points.

Values are instances of Float. The value can be any positive value, although most fonts will pin to a reasonable range.

o **TRANSFORM**

```
public static final String TRANSFORM
```

The transform of the font.

Values are instances of Transform. A null value indicates the font should use its default transform.

o **TRANSFORM_DEFAULT**

```
public static final AffineTransform TRANSFORM_DEFAULT
```

The default transform if TRANSFORM is unspecified. This value is null.

See Also:
TRANSFORM

o **FONT**

```
public static final String FONT
```

The font to use to render this text.

Values are instances of Font. A null value indicates that text layout should try to resolve the font as best it can for the given text.

o **BIDI_EMBEDDING**

```
public static final String BIDI_EMBEDDING
```

The embedding level for nested bidirectional runs.

TextLayout assumes this feature has the same value for all text in a paragraph.

Values are instances of Integer, between 0 and 31 inclusive. Values from 0 to 15 represent embeddings. If bit 4 is set (values from 16 to 31) they represent directional overrides. Even values indicate left–to–right, odd indicate right–to–left. Within an embedding, text assumes a base line direction corresponding to the embedding, and the standard bidirectional properties of the text applies. Within an override, all text flows in the given direction, and the standard bidirectional properties of the text are overridden. See the Unicode Standard v. 2.0, section

3–11.

o **BIDI_EMBEDDING_DEFAULT**

```
public static final Integer BIDI_EMBEDDING_DEFAULT
```

The default embedding if BIDI_EMBEDDING is unspecified. This value is 0.

See Also:
BIDI_EMBEDDING

o **BACKGROUND**

```
public static final String BACKGROUND
```

The background color.

Values are instances of Color. Null indicates a transparent background.

o **BACKGROUND_DEFAULT**

```
public static final Color BACKGROUND_DEFAULT
```

The default background color if BACKGROUND is unspecified. This value is null.

See Also:
BACKGROUND

o **FOREGROUND**

```
public static final String FOREGROUND
```

The foreground color.

Values are instances of Color. Null indicates use the ’current’ color in the Graphics.

o **FOREGROUND_DEFAULT**

```
public static final Color FOREGROUND_DEFAULT
```

The default foreground color if FOREGROUND is unspecified. This value is null.

See Also:
FOREGROUND

o **UNDERLINE**

```
public static final String UNDERLINE
```

Adorn the text with underlining.

Values are instances of Boolean.

o **UNDERLINE_DEFAULT**

```
public static final Boolean UNDERLINE_DEFAULT
```

The default underline if UNDERLINE is unspecified. This value is false, indicating no underline.

See Also:
UNDERLINE

o **UNDERLINE_ON**

```
public static final Boolean UNDERLINE_ON
```

Draw an underline. This value is true.

See Also:
UNDERLINE

o **STRIKETHROUGH**

```
public static final String STRIKETHROUGH
```

Adorn the text with strikethrough.

Values are instances of Boolean.

o **STRIKETHROUGH_DEFAULT**

```
public static final Boolean STRIKETHROUGH_DEFAULT
```

The default strikethrough if STRIKETHROUGH is unspecified, the value is false.

See Also:
STRIKETHROUGH

o **STRIKETHROUGH_ON**

```
public static final Boolean STRIKETHROUGH_ON
```

A single strikethrough. This value is true.

See Also:
STRIKETHROUGH

o **SUPERSUBSCRIPT**

```
public static final String SUPERSUBSCRIPT
```

Super and subscripting.

Values are arrays of two floats. The first value is is used to scale the point size, the second computes the offset as a ratio of the point size. For example, when applied to a 12 point font, a value of [.75, −.5] requests a point size of 9, positioned 6 points above the baseline.

o **SUPERSUBSCRIPT_DEFAULT**

```
public static final float SUPERSUBSCRIPT_DEFAULT[]
```

The default supersubscript if SUPERSUBSCRIPT is unspecified. This value is [1, 0].

See Also:
SUPERSUBSCRIPT

o **SUPERSUBSCRIPT_UP1**

```
public static final float SUPERSUBSCRIPT_UP1[]
```

Standard superscript. This value is [0.75, −0.5].

See Also:
SUPERSUBSCRIPT

o **SUPERSUBSCRIPT_DOWN1**

```
public static final float SUPERSUBSCRIPT_DOWN1[]
```

Standard subscript. This value is [0.75, 0.5].

See Also:
SUPERSUBSCRIPT

o **JUSTIFICATION**

```
public static final String JUSTIFICATION
```

Justification for the paragraph.

TextLayout assumes this feature has the same value for all text in a paragraph. Value is a float, extracted as a Double. This is a ratio controlling the space adjustment. For example, if a line's natural advance is 50 pixels, and it is asked to justify to a width of 100 pixels, a ratio of 1.0 results in a justified line with an advance of 100 pixels, and a ratio of 0.5 results in a justified line with an advance of 75 pixels.

o **JUSTIFICATION_DEFAULT**

```
public static final Float JUSTIFICATION_DEFAULT
```

The default justification if JUSTIFICATION is unspecified. This value is 0.0.

See Also:
JUSTIFICATION

o **JUSTIFICATION_FULL**

```
public static final Float JUSTIFICATION_FULL
```

Justify the line to the full requested width. This value is 1.0.

See Also:
JUSTIFICATION

o **RUN_DIRECTION**

```
public static final String RUN_DIRECTION
```

The run direction of the line, either left−to−right or right−to−left. On vertical lines, left−to−right means top−to−bottom.

TextLayout assumes this feature has the same value for all text in a paragraph.

Values are instances of boolean.

o **RUN_DIRECTION_DEFAULT**

```
public static final Boolean RUN_DIRECTION_DEFAULT
```

The default direction if RUN_DIRECTION is unspecified. This value is false, indicating left−to−right.

See Also:
RUN_DIRECTION

o **RUN_DIRECTION_LTR**

```
public static final Boolean RUN_DIRECTION_LTR
```

Left−to−right (top−to−bottom) run direction. This value is false.

See Also:
RUN_DIRECTION

o **RUN_DIRECTION_RTL**

```
public static final Boolean RUN_DIRECTION_RTL
```

Right−to−left (bottom−to−top) run direction. This value is true.

See Also:
RUN_DIRECTION

o **BIDI_NUMERIC**

```
public static final String BIDI_NUMERIC
```

The bidirectional layout control for roman numeral text.

Values are instances of Integer.

o **BIDI_NUMERIC_STRONG_ROMAN**

```
public static final Integer BIDI_NUMERIC_STRONG_ROMAN
```

Interpret roman numerals as roman numerals always.

o **BIDI_NUMERIC_STRONG_ARABIC**

```
public static final Integer BIDI_NUMERIC_STRONG_ARABIC
```

Interpret roman numerals as arabic numerals always.

o **BIDI_NUMERIC_WEAK_ARABIC**

```
public static final Integer BIDI_NUMERIC_WEAK_ARABIC
```

Interpret roman numerals as having arabic line order.

o **BIDI_NUMERIC_DEFAULT**

```
public static final Integer BIDI_NUMERIC_DEFAULT
```

o **BASELINE**

```
public static final String BASELINE
```

Baseline to which to align lines.

TextLayout assumes this feature has the same value for all text in a paragraph. Regardless of the baselines to which characters align, the layout as a whole will align to this line.

Values are instances of Integer.

See Also:
ROMAN_BASELINE, CENTER_BASELINE, HANGING_BASELINE, getBaselineOffsetsFor, getBaselineFor

o **BASELINE_DEFAULT**

```
public static final Integer BASELINE_DEFAULT
```

The default baseline if BASELINE is unspecified. This value is Font.ROMAN_BASELINE.

See Also:
BASELINE

o **BASELINE_ROMAN**

```
public static final Integer BASELINE_ROMAN
```

Use a roman baseline for the line. This value is Font.ROMAN_BASELINE.

See Also:
BASELINE

o **BASELINE_IDEOGRAPHIC**

```
public static final Integer BASELINE_IDEOGRAPHIC
```

Use a center baseline for the line. This value is Font.CENTER_BASELINE.

See Also:
BASELINE

o **BASELINE_HANGING**

```
public static final Integer BASELINE_HANGING
```

Use a hanging baseline for the line. This value is Font.HANGING_BASELINE.

See Also:
BASELINE

o **BASELINE_OFFSETS**

```
public static final String BASELINE_OFFSETS
```

Baseline offsets used by a line.

TextLayout assumes this feature has the same value for all text in a paragraph.

Values are arrays of three floats. If defaulted, TextLayout attempts to find a reasonable value for all text in the paragraph.

o **BASELINE_OFFSETS_DEFAULT**

```
public static final float BASELINE_OFFSETS_DEFAULT[]
```

The default baseline offsets if BASELINE_OFFSETS is unspecified. This value is null.

See Also:
BASELINE_OFFSETS

o **EMPTY**

```
public static final AttributeSet EMPTY
```

An empty attribute set, for convenience.

Constructors

o **TextAttributeSet**

```
public TextAttributeSet()
```

Construct a new mutable attribute set.

o **TextAttributeSet**

```
public TextAttributeSet(AttributeSet source)
```

Construct a new mutable attribute set containing the provided attributes.

Parameters:
source – the attributes to copy into this set.

o **TextAttributeSet**

```
public TextAttributeSet(String name,
                        Object value)
```

Construct a new mutable attribute set containing the provided attribute.

Parameters:

name – the name of the attribute.
value – the value of the attribute.

Methods

o **isEmpty**

```
public boolean isEmpty()
```

Returns true if the set is empty.

o **getSize**

```
public int getSize()
```

Returns the number of attributes.

o **names**

```
public Enumeration names()
```

Returns an enumeration over the names of the attributes in the set. The elements of the enumeration are all Strings.

o **get**

```
public Object get(String name)
```

Returns the value of the attribute with this name, or null if the attribute is not defined.

o **contains**

```
public boolean contains(String name,
                        Object value)
```

Returns true if this set contains this attribute with an equal value, or if the value is null and the attribute is not defined.

o **contains**

```
public boolean contains(AttributeSet rhs)
```

Returns true if this set contains all the attributes with equal values.

o **equals**

```
public boolean equals(Object rhs)
```

Return true if the argument is an attribute set containing the same attributes as this.

Parameters:
rhs – the object which may be an attribute set.
Overrides:
equals in class Object

o **equals**

```
public boolean equals(AttributeSet rhs)
```

Return true if the argument contains the same attributes as this.

Parameters:
rhs – the attribute set to test against.

o **add**

```
public void add(String name,
                Object value)
```

Remove any existing attribute with the same name, and add a new attribute with the given name and value. The value must be immutable, or not mutated by any client.

Parameters:
name – the name of the attribute to add
value – the value of the attribute to add

o **add**

```
public void add(AttributeSet attributes)
```

Remove any existing attributes with the same names, and add the new attributes.

Parameters:
attributes – the set of attributes to add

o **remove**

```
public void remove(String name)
```

Remove any existing attribute with the given name.

Parameters:
name – the name of the attribute to remove

o **remove**

```
public void remove(Enumeration names)
```

Remove any existing attributes with the given names.

Parameters:
names – an enumeration over the names of attributes to remove. The elements of the enumeration are Strings.

o **set**

```
public void set(AttributeSet attributes)
```

Removes all attributes, then adds the new attributes.

o **clone**

```
public Object clone()
```

Return a copy of this attribute set. Attributes are shared with the copy.

Overrides:
clone in class Object

Class java.awt.font.TextHitInfo

```
java.lang.Object
   |
   +----java.awt.font.TextHitInfo
```

public final class **TextHitInfo**
extends Object

This class represents hit test information for characters in a TextLayout. This information enables developers to correctly draw a caret identifying an insertion point when a user clicks on a point in the bounding box of a TextLayout,

Text has direction. In bi−directional text, there is both left−to−right and right−to−left direction. In Roman text *left* of the glyph is called the leading edge and *right* is the trailing edge. However, in Hebrew or Arabic, the right of the glyph is the leading edge and the left is the trailing edge.

In a bi−directional string, it is important to know if the glyph is at the boundary of left−to−right glyphs and right−to−left glyphs. If that is true, then isLeadingEdge() indicates whether the next insertion is a left−to−right direction or a right−to−left direction.

See Also:
TextLayout, hitTestChar

Variable Index

o **LEADING**
o **TRAILING**

Constructor Index

o **TextHitInfo**(int, boolean)
Constructor for TextHitInfo.

Method Index

o **equals**(Object)
o **getCharIndex**()
　　Returns the index of the character where the user has clicked.
o **getInsertionIndex**()
　　Returns the insertion index.
o **getOffsetHit**(int)
　　Returns a hit whose character index is offset by delta.
o **getOtherHit**()
　　Returns the hit on the other side of the insertion point from hit.
o **hashCode**()
　　Returns the hash code.
o **isLeadingEdge**()
　　Returns true if the hit is on the leading edge of a glyph and false if the hit is on the trailing edge.
o **leading**(int)
　　Create a hit on the leading edge of the character at charIndex.
o **toString**()
o **trailing**(int)
　　Create a hit on the trailing edge of the character at charIndex.

Variables

o **LEADING**

```
public static boolean LEADING
```

o **TRAILING**

```
public static boolean TRAILING
```

Constructors

o **TextHitInfo**

```
public TextHitInfo(int charIndex,
                   boolean isLeadingEdge)
```

Constructor for TextHitInfo.

Parameters:
charIndex – identifies the character index in the TextLayout.
isLeadingEdge – true if the hit is on the leading edge of a glyph.

Methods

o **getCharIndex**

```
public int getCharIndex()
```

Returns the index of the character where the user has clicked.

o **isLeadingEdge**

```
public boolean isLeadingEdge()
```

Returns true if the hit is on the leading edge of a glyph and false if the hit is on the trailing edge.

o **getInsertionIndex**

```
public int getInsertionIndex()
```

Returns the insertion index. This is the character index if the leading edge of the character was hit, and one greater than the character index if the trailing edge was hit.

o **hashCode**

```
public int hashCode()
```

Returns the hash code.

Overrides:
hashCode in class Object

o **equals**

```
public boolean equals(Object obj)
```

Overrides:
equals in class Object

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

o **leading**

```
public static TextHitInfo leading(int charIndex)
```

Create a hit on the leading edge of the character at charIndex.

o **trailing**

```
public static TextHitInfo trailing(int charIndex)
```

Create a hit on the trailing edge of the character at charIndex.

o **getOtherHit**

```
public TextHitInfo getOtherHit()
```

Returns the hit on the other side of the insertion point from hit.

o **getOffsetHit**

```
public TextHitInfo getOffsetHit(int delta)
```

Returns a hit whose character index is offset by delta.

Class java.awt.font.TextLayout

```
java.lang.Object
   |
   +----java.awt.font.TextLayout
```

---

public final class **TextLayout**
extends Object
implements Cloneable

This class encapsulates extensions to the layout information for drawing text. It also provides information for hit testing of text.

See Also:
drawString

Constructor Index

- o **TextLayout**(AttributedCharacterIterator)
  Construct a layout from an iterator over styled text.
- o **TextLayout**(String, Font)
  Construct a layout from a string and a font.

Method Index

- o **clone**()
  Create a copy of this layout.
- o **deleteChar**(AttributedCharacterIterator, int, float)
  An optimization to facilitate deleting single characters from left−to−right text.
- o **draw**(Graphics2D, float, float)
  Render the layout at the provided location in the graphics.
- o **equals**(Object)
  Return true if the object is a TextLayout and this equals the object.
- o **equals**(TextLayout)
  Return true if the two layouts are equal.
- o **getAdvance**()

Return the advance of the layout.

o **getAllAvailableAttributes**()
    Return list of the attributes used by this layout.
o **getAscent**()
    Return the ascent of the layout.
o **getBaseline**()
    Return the baseline for this layout.
o **getBaselineOffsets**()
    Return the offsets array for the baselines used for this layout.
o **getBlackBoxBounds**(int, int)
    Return the black box bounds of the characters between start and limit.
o **getBounds**()
    Return the bounds of the layout.
o **getCaretInfo**(TextHitInfo)
    Return information about the caret corresponding to hit.
o **getCaretShapes**(TextHitInfo)
    A convenience override that uses the natural bounds of the layout as the bounds.
o **getCaretShapes**(TextHitInfo, Rectangle2D)
    Return two paths corresponding to the strong and weak caret.
o **getCharacterCount**()
    Return the number of characters represented by this layout.
o **getDescent**()
    Return the descent of the layout.
o **getHighlightShape**(TextHitInfo, TextHitInfo, boolean)
    A convenience overload that uses the natural bounds of the text as the bounds used to determine the highlight path.
o **getHighlightShape**(TextHitInfo, TextHitInfo, boolean, Rectangle2D)
    Return a logical or visual selection corresponding to startHit and limitHit.
o **getJustifiedLayout**(float)
    Create a copy of this layout justified to the given width.
o **getLeading**()
    Return the leading of the layout.
o **getLineBreakIndex**(int, float)
    Accumulate the advances of the characters at and after startPos, until a character is reached whose advance would equal or exceed width.
o **getLogicalHighlightShape**(int, int, Rectangle2D)
    Return a path enclosing the logical selection between start and limit, extended to bounds.
o **getLogicalRangesForVisualSelection**(TextHitInfo, TextHitInfo)
    Return the logical ranges of text corresponding to a visual selection.
o **getNextLeftHit**(TextHitInfo)
    Return the hit for the next strong caret to the left (top); if no such hit, return null.
o **getNextRightHit**(TextHitInfo)
    Return the hit for the next strong caret to the right (bottom); if no such hit, return null.
o **getVisibleAdvance**()
    Return the advance of the layout, minus trailing whitespace.
o **getVisualHighlightShape**(TextHitInfo, TextHitInfo, Rectangle2D)
    Return a path enclosing the visual selection between start and limit, extended to bounds.
o **handleJustify**(float)
    Justify this layout.

o **hashCode**()
    Return the hash code of this layout.
o **hitTestChar**(float, float)
    Return a TextHitInfo corresponding to the point.
o **insertChar**(AttributedCharacterIterator, int, float)
    An optimization to facilitate inserting single characters into left–to–right text.
o **isLeftToRight**()
    Return true if the layout is left–to–right.
o **isVertical**()
    Return true if the layout is vertical.
o **sublayout**(int, int)
    Return a layout that represents a subsection of this layout.
o **toString**()
    Display the glyphsets contained in the layout, for debugging only.

Constructors

o **TextLayout**

```
public TextLayout(String string,
                  Font font)
```

Construct a layout from a string and a font. All the text is styled using the provided font. The string must specify a single paragraph of text, as an entire paragraph is required for the bidi algorithm. Lines may then be constructed by taking subranges of the layout.

Parameters:
str – the text to display.
font – a font used to style the text.

o **TextLayout**

```
public TextLayout(AttributedCharacterIterator text)
```

Construct a layout from an iterator over styled text. The iterator must specify a single paragraph of text, as an entire paragraph is required for the bidi algorithm. Lines may then be constructed by taking subranges of the layout.

Parameters:
text – the styled text to display.

Methods

o **clone**

```
protected Object clone()
```

Create a copy of this layout.

Overrides:

clone in class Object

o **getJustifiedLayout**

```
public TextLayout getJustifiedLayout(float justificationWidth)
```

Create a copy of this layout justified to the given width. If this layout has itself already been justified, or the justification ratio is zero, return this layout unchanged.

Parameters:

justificationWidth − the width to use when justifying the line. For best results, it should not be too different from the current advance of the line.

Returns:

a layout justified to the given width.

See Also:

justify

o **handleJustify**

```
protected void handleJustify(float justificationWidth)
```

Justify this layout. Overridden by subclassers to control justification. The layout will only justify if the paragraph attributes (from the source text, possibly defaulted by the layout attributes) indicate a non−zero justification ratio. The text will be justified to the indicated width. The current implementation also adjusts hanging punctuation and trailing whitespace to overhang the justification width. Once justified, the layout may not be rejustified.

Some code may rely on immutablity of layouts. Subclassers should not call this directly, but instead should call getJustifiedLayout, which will call this method on a clone of this layout, preserving the original.

Parameters:

justificationWidth − the width to use when justifying the line. For best results, it should not be too different from the current advance of the line.

See Also:

getJustifiedLayout

o **getBaseline**

```
public int getBaseline()
```

Return the baseline for this layout. The baseline is one of the values defined in Font (roman, centered, hanging). Ascent and descent are relative to this baseline. The baselineOffsets are also relative to this baseline.

See Also:
getBaselineOffsets, Font

o **getBaselineOffsets**

```
public float[] getBaselineOffsets()
```

Return the offsets array for the baselines used for this layout. The array is indexed by one of the values defined in Font (roman, centered, hanging). The values are relative to this layout's baseline, so that getBaselineOffsets[getBaseline()] == 0. Offsets are added to the position of the layout's baseline to get the position for the new baseline, i.e. on horizontal layouts, negative offsets are up, and on vertical layouts, negative offsets are left.
!!! This differs from other metric information, where on vertical lines the math works as though x positions to the right were lower instead of higher.

See Also:
getBaseline, Font

o **getAdvance**

```
public float getAdvance()
```

Return the advance of the layout. The advance is the distance from the origin to the advance of the rightmost (bottommost) character measuring in the line direction.

o **getVisibleAdvance**

```
public float getVisibleAdvance()
```

Return the advance of the layout, minus trailing whitespace.

See Also:
getAdvance

o **getAscent**

```
public float getAscent()
```

Return the ascent of the layout. The ascent is the distance from the top (right) of the layout to the baseline. It is always positive or zero. The ascent is sufficient to accomodate superscripted text and is the maximum of the sum of the the ascent, offset, and baseline of each glyph.

o **getDescent**

```
public float getDescent()
```

Return the descent of the layout. The descent is the distance from the baseline to the bottom (left) of the layout. It is always positive or zero. The descent is sufficient to accomodate subscripted text and is maximum of the sum of the descent, offset, and baseline of each glyph.

o **getLeading**

```
public float getLeading()
```

Return the leading of the layout. The leading is the suggested interline spacing for this layout.

o **getBounds**

```
public Rectangle2D getBounds()
```

Return the bounds of the layout. This bounds all space that can be drawn on by text or carets. It extends from ascent to descent + leading, and from the left (top) edge of the first character to the right (bottom) edge of the last. If there is hanging punctuation, the left (top) of the bounds may be negative.

o **isLeftToRight**

```
public boolean isLeftToRight()
```

Return true if the layout is left−to−right. The layout has a base direction of either left−to−right (LTR) or right−to−left (RTL). This is independent of the actual direction of text on the line, which may be either or mixed. Left−to−right layouts by default should position flush left, and if on a tabbed line, the tabs run left to right, so that logically successive layouts position left to right. The opposite is true for RTL layouts. By default they should position flush left, and tabs run right−to−left. On vertical lines all text runs top−to−bottom, and is treated as LTR.

o **isVertical**

```
public boolean isVertical()
```

Return true if the layout is vertical.

o **getCharacterCount**

```
public int getCharacterCount()
```

Return the number of characters represented by this layout.

o **getCaretInfo**

```
public float[] getCaretInfo(TextHitInfo hit)
```

Return information about the caret corresponding to hit. The first element of the array is the intersection of the caret with the baseline. The second element of the array is the slope (run/rise) of the caret.

Parameters:
: hit – a hit on a character in this layout

Returns:
: a two–element array containing the position and slope of the caret

o **getNextRightHit**

```
public TextHitInfo getNextRightHit(TextHitInfo hit)
```

Return the hit for the next strong caret to the right (bottom); if no such hit, return null. If the hit character index is out of bounds, an IllegalArgumentException is thrown.

Parameters:
: hit – a hit on a character in this layout.

Returns:
: a hit whose strong caret appears at the next position to the right (bottom) of the strong caret of the provided hit, or null.

o **getNextLeftHit**

```
public TextHitInfo getNextLeftHit(TextHitInfo hit)
```

Return the hit for the next strong caret to the left (top); if no such hit, return null. If the hit character index is out of bounds, an IllegalArgumentException is thrown.

Parameters:
: hit – a hit on a character in this layout.

Returns:
: a hit whose strong caret appears at the next position to the left (top) of the strong caret of the provided hit, or null.

o **getCaretShapes**

```
public Shape[] getCaretShapes(TextHitInfo hit,
                              Rectangle2D bounds)
```

Return two paths corresponding to the strong and weak caret.

Parameters:
: hit – a hit on a character in this layout
: bounds – the bounds to which to extend the carets

Returns:
: an array of two paths. Element zero is the strong caret. There is always a strong caret. Element one is the weak caret (if any) or null.

o **getCaretShapes**

```
public Shape[] getCaretShapes(TextHitInfo hit)
```

A convenience override that uses the natural bounds of the layout as the bounds.

o **getHighlightShape**

```
public Area getHighlightShape(TextHitInfo startHit,
                              TextHitInfo limitHit,
                              boolean visual,
                              Rectangle2D bounds)
```

Return a logical or visual selection corresponding to startHit and limitHit.

Parameters:
startHit – a hit representing the start of the selection.
limitHit – a hit representing the limit of the selection.
visual – true if the selection should be visually contiguous. The area between the carets defined by startHit and limitHit will be selected. Otherwise, the hits are mapped to insertion points, and the characters between these insertion points will be selected. In mixed directional text, this will result in a discontiguous selection (representing the contiguous logical ranges of text).
bounds – the bounds of the path. On horizontal lines, the top and bottom of the bounds determine the top and bottom of the resulting path, and the right and left are used to extend selections that include the characters on the endpoints of the line. On vertical lines the roles of top/bottom and left/right are exchanged. The origin of bounds is at position 0,0 in the layout.

See Also:
getLogicalHighlightShape, getVisualHighlightShape

o **getHighlightShape**

```
public Area getHighlightShape(TextHitInfo startHit,
                              TextHitInfo limitHit,
                              boolean visual)
```

A convenience overload that uses the natural bounds of the text as the bounds used to determine the highlight path.

o **getLogicalRangesForVisualSelection**

```
public int[] getLogicalRangesForVisualSelection(TextHitInfo startHit,
                                                TextHitInfo limitHit)
```

Return the logical ranges of text corresponding to a visual selection.

Parameters:
startHit – the start of the visual range
limitHit – the end of the visual range

Returns:
an array of integers representing start/limit pairs for the selected ranges

See Also:
getVisualHighlightShape

o **getVisualHighlightShape**

```
public Area getVisualHighlightShape(TextHitInfo startHit,
                                    TextHitInfo limitHit,
                                    Rectangle2D bounds)
```

Return a path enclosing the visual selection between start and limit, extended to bounds.

If the selection includes the leftmost (topmost) position, the selection is extended to the left (top) of the bounds. If the selection includes the rightmost (bottommost) position, the selection is extended to the right (bottom) of the bounds. The height (width on vertical lines) of the selection is always extended to bounds.

Although the selection is always contiguous, the logically selected text can be discontiguous on lines with mixed−direction text. The logical ranges of text selected can be retrieved using getLogicalRangesForVisualSelection. For example, consider the text 'ABCdef' where capital letters indicate right−to−left text, rendered on a right−to−left line, with a visual selection from 0L (the leading edge of 'A') to 3T (the trailing edge of 'd'). The text appears as follows, with bold underlined areas representing the selection:

d**efCBA**

The logical selection ranges are 0−3, 4−6 (ABC, ef) because the visually contiguous text is logically discontiguous. Also note that since the rightmost position on the layout (to the right of 'A') is selected, the selection is extended to the right of the bounds.

Parameters:
startHit − the start of the visual selection
limitHit − the limit of the visual selection
bounds − the bounding rectangle to which to extend the selection
Returns:
an area enclosing the selection
See Also:
getLogicalRangesForVisualSelection, getLogicalHighlightShape

o **getLogicalHighlightShape**

```
public Area getLogicalHighlightShape(int start,
                                     int limit,
                                     Rectangle2D bounds)
```

Return a path enclosing the logical selection between start and limit, extended to bounds.

If the selection range includes the first logical character, the selection is extended to the portion of bounds before the start of the layout. If the range includes the last logical character, the selection is extended to the portion of bounds after the end of the layout. The height (width on vertical lines) of the selection is always extended to bounds.

The selection can be discontiguous on lines with mixed−direction text. Only those characters in the logical range between start and limit will appear selected. For example consider the text 'ABCdef' where capital letters indicate right−to−left text, rendered on a right−to−left line, with a logical selection from 0 to 4 ('ABCd'). The text appears as follows, with bold standing in for the selection, and underlining for the extension:

```
defCBA
```

The selection is discontiguous because the selected characters are visually discontiguous. Also note that since the range includes the first logical character (A), the selection is extended to the portion of the bounds before the start of the layout, which in this case (a right−to−left line) is the right portion of the bounds.

Parameters:
start − the start of the range of characters to select
limit − the limit of the range of characters to select
bounds − the bounding rectangle to which to extend the selection
Returns:
an area enclosing the selection
See Also:
getVisualHighlightShape

o **getBlackBoxBounds**

```
public Area getBlackBoxBounds(int start,
                              int limit)
```

Return the black box bounds of the characters between start and limit. The black box bounds is an area consisting of the union of the bounding boxes of all the glyphs corresponding to the characters between start and limit. This path may be disjoint.

Parameters:
start − the index of the first character to include in the bounds
limit − the index past the last character to include in the bounds
Returns:
a path enclosing the black box bounds

o **getLineBreakIndex**

```
public int getLineBreakIndex(int startPos,
                             float width)
```

Accumulate the advances of the characters at and after startPos, until a character is reached whose advance would equal or exceed width. Return the index of that character.

o **hitTestChar**

```
public TextHitInfo hitTestChar(float x,
```

```
                                float y)
```

Return a TextHitInfo corresponding to the point. Coordinates outside the bounds of the layout map to hits on the leading edge of the first logical character, or the trailing edge of the last logical character, as appropriate, regardless of the position of that character in the line. Only the direction along the baseline is used to make this evaluation.

Parameters:
: x – the x offset from the origin of the layout
: y – the y offset from the origin of the layout

Returns:
: a hit describing the character and edge (leading or trailing) under the point

o **getAllAvailableAttributes**

```
public static String[] getAllAvailableAttributes()
```

Return list of the attributes used by this layout.

o **sublayout**

```
public TextLayout sublayout(int start,
                            int limit)
```

Return a layout that represents a subsection of this layout. The number of characters must be >= 1. The original layout must not be justified. The new layout will apply the bidi 'line reordering' rules to the text.

Parameters:
: start – the index of the first character to use
: limit – the index past the last character to use

Returns:
: a new layout

o **insertChar**

```
public TextLayout insertChar(AttributedCharacterIterator newText,
                             int insertPos,
                             float maxAdvance)
```

An optimization to facilitate inserting single characters into left–to–right text. This attempts to produce a new layout based on modifications to this one. If that cannot be be done easily, this returns null. Clients should always be prepared to handle a null result. If null is returned, the client should construct a new layout.

Parameters:
: newText – the text for the new layout. This represents the text after the insertion occurred. The new layout will represent all the text.
: insertPos – the position at which the single character was inserted.

maxAdvance – a maximum advance the new layout must fit.

Returns:

a new layout representing newText whose advance is less than maxAdvance, or null.

o **deleteChar**

```
public TextLayout deleteChar(AttributedCharacterIterator newText,
                             int deletePos,
                             float minAdvance)
```

An optimization to facilitate deleting single characters from left–to–right text. This attempts to produce a new layout based on modifications to this one. If that cannot be be done easily, this returns null. Clients should always be prepared to handle a null result. If null is returned, the client should construct a new layout.

Parameters:

newText – the text for the new layout. This represents the text after the deletion occurred. The new layout will represent all the text.
deletePos – the position at which the single character was deleted.
minAdvance – a minimum advance the new layout must fit.

Returns:

a new layout representing newText whose advance equals or exceeds minAdvance, or null.

o **hashCode**

```
public int hashCode()
```

Return the hash code of this layout.

Overrides:

hashCode in class Object

o **equals**

```
public boolean equals(Object obj)
```

Return true if the object is a TextLayout and this equals the object.

Overrides:

equals in class Object

o **equals**

```
public boolean equals(TextLayout layout)
```

Return true if the two layouts are equal. Two layouts are equal if they contain equal glyphsets in the same order.

Parameters:
: layout – the layout to which to compare this layout.

o **toString**

```
public String toString()
```

Display the glyphsets contained in the layout, for debugging only.

Overrides:
: toString in class Object

o **draw**

```
public void draw(Graphics2D g2,
                 float x,
                 float y)
```

Render the layout at the provided location in the graphics. The origin of the layout is placed at x, y. Rendering may touch any point within getBounds() of this position. This leaves the graphics unchanged.

Parameters:
: g2 – the graphics into which to render the layout
: x – the x position for the origin of the layout
: y – the y position for the origin of the layout

See Also:
: getBounds

Interface java.awt.geom.PathIterator

public interface **PathIterator**

Variable Index

o **SEG_CLOSE**
The point type that specifies that the preceding subpath should be closed by appending a line segment back to the point corresponding to the most recent SEG_MOVETO.
o **SEG_CUBICTO**
The point type for the set of 3 points that specify a cubic parametric curve to be drawn from the most recently specified point.
o **SEG_LINETO**
The point type for a point that specifies the end point of a line to be drawn from the most recently specified point.
o **SEG_MOVETO**
The point type for a point that specifies the starting location for a new subpath.
o **SEG_QUADTO**
The point type for the pair of points that specify a quadratic parametric curve to be drawn from the most recently specified point.
o **WIND_EVEN_ODD**
An even−odd winding rule for determining the interior of a path.
o **WIND_NON_ZERO**
A non−zero winding rule for determining the interior of a path.

Method Index

o **currentSegment**(double[])
Returns the coordinates and type of the current path segment in the iteration.
o **currentSegment**(float[])
Returns the coordinates and type of the current path segment in the iteration.
o **getWindingRule**()
Return the winding rule for determining the interior of the path.
o **isDone**()
Tests if there are more points to read.
o **next**()
Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction.

Variables

o **WIND_EVEN_ODD**

```
public static final byte WIND_EVEN_ODD
```

An even−odd winding rule for determining the interior of a path.

o **WIND_NON_ZERO**

```
public static final byte WIND_NON_ZERO
```

A non−zero winding rule for determining the interior of a path.

o **SEG_MOVETO**

```
public static final byte SEG_MOVETO
```

The point type for a point that specifies the starting location for a new subpath.

o **SEG_LINETO**

```
public static final byte SEG_LINETO
```

The point type for a point that specifies the end point of a line to be drawn from the most recently specified point.

o **SEG_QUADTO**

```
public static final byte SEG_QUADTO
```

The point type for the pair of points that specify a quadratic parametric curve to be drawn from the most recently specified point. The curve is interpolated by solving the parametric control equation in the range `(t = [0..1])` using the most recently specified (current) point (CP), the first control point (P1), and the final interpolated control point (P2). The parametric control equation for this curve is:

```
  P(t) = B(2,0)*CP + B(2,1)*P1 + B(2,2)*P2
  0 <= t <= 1
B(n,m) = mth coefficient of nth degree Bernstein polynomial
       = C(n,m) * t^(m) * (1 - t)^(n-m)
C(n,m) = Combinations of n things, taken m at a time
       = n! / (m! * (n-m)!)
```

o **SEG_CUBICTO**

```
public static final byte SEG_CUBICTO
```

The point type for the set of 3 points that specify a cubic parametric curve to be drawn from the most recently specified point. The curve is interpolated by solving the parametric control equation in the range `(t = [0..1])` using the most recently specified (current) point (CP), the first control point (P1), the second control point (P2), and the final interpolated control point (P3). The parametric control equation for this curve is:

```
    P(t) = B(3,0)*CP + B(3,1)*P1 + B(3,2)*P2 + B(3,3)*P3
    0 <= t <= 1
  B(n,m) = mth coefficient of nth degree Bernstein polynomial
         = C(n,m) * t^(m) * (1 - t)^(n-m)
  C(n,m) = Combinations of n things, taken m at a time
         = n! / (m! * (n-m)!)
```

This form of curve is commonly known as a Bézier curve.

o **SEG_CLOSE**

```
public static final byte SEG_CLOSE
```

The point type that specifies that the preceding subpath should be closed by appending a line segment back to the point corresponding to the most recent SEG_MOVETO.

Methods

o **getWindingRule**

```
public abstract int getWindingRule()
```

Return the winding rule for determining the interior of the path.

See Also:
WIND_EVEN_ODD, WIND_NON_ZERO

o **isDone**

```
public abstract boolean isDone()
```

Tests if there are more points to read.

Returns:
true if there are more points to read

o **next**

```
public abstract void next()
```

Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction.

o **currentSegment**

```
public abstract int currentSegment(float coords[])
```

Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, or SEG_CLOSE. A float array of length 6 must be passed in and may be used to store the coordinates of the point(s). Each point is stored as a pair of float x,y coordinates. SEG_MOVETO and SEG_LINETO types will return one point, SEG_QUADTO will return two points, SEG_CUBICTO will return 3 points and SEG_CLOSE will not return any points.

See Also:
SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, SEG_CLOSE

o **currentSegment**

```
public abstract int currentSegment(double coords[])
```

Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, or SEG_CLOSE. A double array of length 6 must be passed in and may be used to store the coordinates of the point(s). Each point is stored as a pair of double x,y coordinates. SEG_MOVETO and SEG_LINETO types will return one point, SEG_QUADTO will return two points, SEG_CUBICTO will return 3 points and SEG_CLOSE will not return any points.

See Also:
SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, SEG_CLOSE

Class java.awt.geom.AffineTransform

```
java.lang.Object
   |
   +----java.awt.geom.AffineTransform
```

public class **AffineTransform**
extends Object
implements Cloneable

This class represents a 2D affine transform which performs a linear mapping from 2D coordinates to other 2D coordinates in a manner which preserves the "straightness" and "parallelness" of lines. Affine transformations can be constructed using sequences of translations, scales, flips, rotations, and shears.

Such a coordinate transformation can be represented by a 3 row by 3 column matrix with an implied last row of [0 0 1] which is applied to the coordinates by considering them to be a column vector and multiplying the coordinate vector by the matrix according to the following process.

```
[  m00  m01  m02  ] [ x ]   [ m00x + m01y + m02 ]
[  m10  m11  m12  ] [ y ] = [ m10x + m11y + m12 ]
[   0    0    1   ] [ 1 ]   [         1         ]
```

Variable Index

o **ANY_ROTATION_MASK**
  This flag is a bit mask for any of the rotation flag bits.
o **ANY_SCALE_MASK**
  This flag is a bit mask for any of the scale flag bits.
o **GENERAL_ROTATION**
  This flag bit indicates that the transform defined by this object performs a rotation by an arbitrary angle in addition to the conversions indicated by other flag bits.
o **GENERAL_SCALE**
  This flag bit indicates that the transform defined by this object performs a general scale in addition to the conversions indicated by other flag bits.
o **GENERAL_TRANSFORM**
  This flag bit indicates that the transform defined by this object performs an arbitrary conversion of the input coordinates.

o **IDENTITY**

This constant indicates that the transform defined by this object is an identity transform.

o **QUADRANT_ROTATION**

This flag bit indicates that the transform defined by this object performs a quadrant rotation by some multiple of 90 degrees in addition to the conversions indicated by other flag bits.

o **TRANSLATION**

This flag bit indicates that the transform defined by this object performs a translation in addition to the conversions indicated by other flag bits.

o **UNIFORM_SCALE**

This flag bit indicates that the transform defined by this object performs a uniform scale in addition to the conversions indicated by other flag bits.

Constructor Index

o **AffineTransform**()

Constructs a new AffineTransform representing the Identity transformation.

o **AffineTransform**(double, double, double, double, double, double)

Constructs a new AffineTransform from 6 double precision values representing the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**(double[])

Constructs a new AffineTransform from an array of double precision values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**(double[][])

Constructs a new AffineTransform from a 2−dimensional array of double precision values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**(float, float, float, float, float, float)

Constructs a new AffineTransform from 6 floating point values representing the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**(float[])

Constructs a new AffineTransform from an array of floating point values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**(float[][])

Constructs a new AffineTransform from a 2−dimensional array of floating point values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix.

Method Index

o **append**(AffineTransform)

Deprecated.

o **clone**()

Returns a copy of this AffineTransform object.

o **concatenate**(AffineTransform)

Concatenates an AffineTransform Tx to this AffineTransform Cx in the most commonly useful way to provide a new user space which is mapped to the former user space by Tx.

o **createInverse**()

Returns a Transform object representing the inverse transformation.

o **createTransformedShape**(Shape)

Transforms a shape object by this transform.

o **equals**(Object)

o **getMatrix**(double[])

Retrieves the 6 specifiable values in the 3x3 affine transformation matrix into an array of double precisions values.

o **getMatrix**(double[][])

Retrieves the 6 specifiable values in the 3x3 affine transformation matrix into a 2−dimensional array of double precision values.

o **getScaleX**()

Returns the X coordinate scaling element (m00) of the 3x3 affine transformation matrix.

o **getScaleY**()

Returns the Y coordinate scaling element (m11) of the 3x3 affine transformation matrix.

o **getShearX**()

Returns the X coordinate shearing element (m01) of the 3x3 affine transformation matrix.

o **getShearY**()

Returns the Y coordinate shearing element (m10) of the 3x3 affine transformation matrix.

o **getTranslateX**()

Returns the X coordinate translation element (m02) of the 3x3 affine transformation matrix.

o **getTranslateY**()

Returns the Y coordinate translation element (m12) of the 3x3 affine transformation matrix.

o **getType**()

Retrieves the flag bits describing the conversion properties of this transform.

o **inverseTransform**(double[], int, double[], int, int)

Inverse transforms an array of double precision coordinates by this transform.

o **inverseTransform**(Point2D, Point2D)

Inverse transforms the specified ptSrc and stores the result in ptDst.

o **isIdentity**()

Returns the boolean true value if this AffineTransform is an identity transform.

o **preConcatenate**(AffineTransform)

Concatenates an AffineTransform Tx to this AffineTransform Cx in a less commonly used way such that Tx modifies the coordinate transformation relative to the absolute pixel space rather than relative to the existing user space.

o **prepend**(AffineTransform)

Deprecated.

o **rotate**(double)

Concatenates this transform with a rotation transformation.

o **rotate**(double, double, double)

Concatenates this transform with a translated rotation transformation.

o **scale**(double, double)

Concatenates this transform with a scaling transformation.

o **setToIdentity**()

Resets this transform to the Identity transform.

o **setToRotation**(double)

Sets this transform to a rotation transformation.

o **setToRotation**(double, double, double)

Sets this transform to a translated rotation transformation.

o **setToScale**(double, double)

Sets this transform to a scaling transformation.
o **setToShear**(double, double)
Sets this transform to a shearing transformation.
o **setToTranslation**(double, double)
Sets this transform to a translation transformation.
o **shear**(double, double)
Concatenates this transform with a shearing transformation.
o **toString**()
Returns a String that represents the value of this Object.
o **transform**(double[], int, double[], int, int)
Transforms an array of double precision coordinates by this transform.
o **transform**(double[], int, float[], int, int)
Transforms an array of double precision coordinates by this transform, storing the results into an array of floats.
o **transform**(float[], int, double[], int, int)
Transforms an array of floating point coordinates by this transform, storing the results into an array of doubles.
o **transform**(float[], int, float[], int, int)
Transforms an array of floating point coordinates by this transform.
o **transform**(Point2D, Point2D)
Transforms the specified ptSrc and stores the result in ptDst.
o **transform**(Point2D[], int, Point2D[], int, int)
Transforms an array of point objects by this transform.
o **translate**(double, double)
Concatenates this transform with a translation transformation.

Variables

o **IDENTITY**

```
public static final int IDENTITY
```

This constant indicates that the transform defined by this object is an identity transform. An identity transform is one in which the output coordinates are always the same as the input coordinates. If this transform is anything other than the identity transform, the type will be a combination of the appropriate flag bits for the various coordinate conversions that this transform performs.

See Also:
TRANSLATION, UNIFORM_SCALE, GENERAL_SCALE, QUADRANT_ROTATION, GENERAL_ROTATION, GENERAL_TRANSFORM, getType

o **TRANSLATION**

```
public static final int TRANSLATION
```

This flag bit indicates that the transform defined by this object performs a translation in addition to the conversions indicated by other flag bits. A translation moves the coordinates by a constant amount in x and y without changing the length or angle of vectors.

See Also:
IDENTITY, UNIFORM_SCALE, GENERAL_SCALE, QUADRANT_ROTATION, GENERAL_ROTATION, GENERAL_TRANSFORM, getType

o **UNIFORM_SCALE**

```
public static final int UNIFORM_SCALE
```

This flag bit indicates that the transform defined by this object performs a uniform scale in addition to the conversions indicated by other flag bits. A uniform scale multiplies the length of vectors by the same amount in both the x and y directions without changing the angle between vectors. This flag bit is mutually exclusive with the GENERAL_SCALE flag.

See Also:
IDENTITY, TRANSLATION, GENERAL_SCALE, QUADRANT_ROTATION, GENERAL_ROTATION, GENERAL_TRANSFORM, getType

o **GENERAL_SCALE**

```
public static final int GENERAL_SCALE
```

This flag bit indicates that the transform defined by this object performs a general scale in addition to the conversions indicated by other flag bits. A general scale multiplies the length of vectors by different amounts in the x and y directions without changing the angle between vectors. This flag bit is mutually exclusive with the UNIFORM_SCALE flag.

See Also:
IDENTITY, TRANSLATION, UNIFORM_SCALE, QUADRANT_ROTATION, GENERAL_ROTATION, GENERAL_TRANSFORM, getType

o **ANY_SCALE_MASK**

```
public static final int ANY_SCALE_MASK
```

This flag is a bit mask for any of the scale flag bits.

o **QUADRANT_ROTATION**

```
public static final int QUADRANT_ROTATION
```

This flag bit indicates that the transform defined by this object performs a quadrant rotation by some multiple of 90 degrees in addition to the conversions indicated by other flag bits. A rotation changes the angles of vectors by the same amount regardless of the original direction of the vector without changing the length of the vector. This flag bit is mutually exclusive with the GENERAL_ROTATION flag.

See Also:
IDENTITY, TRANSLATION, UNIFORM_SCALE, GENERAL_SCALE, GENERAL_ROTATION, GENERAL_TRANSFORM, getType

o **GENERAL_ROTATION**

```
public static final int GENERAL_ROTATION
```

This flag bit indicates that the transform defined by this object performs a rotation by an arbitrary angle in addition to the conversions indicated by other flag bits. A rotation changes the angles of vectors by the same amount regardless of the original direction of the vector without changing the length of the vector. This flag bit is mutually exclusive with the QUADRANT_ROTATION flag.

See Also:
IDENTITY, TRANSLATION, UNIFORM_SCALE, GENERAL_SCALE, QUADRANT_ROTATION, GENERAL_TRANSFORM, getType

o **ANY_ROTATION_MASK**

```
public static final int ANY_ROTATION_MASK
```

This flag is a bit mask for any of the rotation flag bits.

o **GENERAL_TRANSFORM**

```
public static final int GENERAL_TRANSFORM
```

This flag bit indicates that the transform defined by this object performs an arbitrary conversion of the input coordinates. This flag bit is mutually exclusive with any of the other flags.

See Also:
IDENTITY, TRANSLATION, UNIFORM_SCALE, GENERAL_SCALE, QUADRANT_ROTATION, GENERAL_ROTATION, getType

Constructors

o **AffineTransform**

```
public AffineTransform()
```

Constructs a new AffineTransform representing the Identity transformation.

o **AffineTransform**

```
public AffineTransform(float m00,
                       float m10,
                       float m01,
```

```
                        float m11,
                        float m02,
                        float m12)
```

Constructs a new AffineTransform from 6 floating point values representing the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**

```
public AffineTransform(float flatmatrix[])
```

Constructs a new AffineTransform from an array of floating point values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix. The values are retrieved from the array as { m00 m10 m01 m11 [m02 m12] }.

Parameters:

flatmatrix − The float array containing the values to be set in the new AffineTransform object. The length of the array is assumed to be at least 4. If the length of the array is less than 6, only the first 4 values are taken. If the length of the array is greater than 6, the first 6 values are taken.

o **AffineTransform**

```
public AffineTransform(float matrix[][])
```

Constructs a new AffineTransform from a 2−dimensional array of floating point values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix. The values are retrieved from the 2−dimensional array using the row index as the first subscript and the column index as the second.

Parameters:

matrix − The matrix used to construct the new AffineTransform object. Matrix is assumed to be at least 2x2. If the matrix is smaller than 2x3, only the left upper 2x2 matrix is taken. If the matrix is larger than 2x3, the left upper 2x3 matrix is taken.

o **AffineTransform**

```
public AffineTransform(double m00,
                       double m10,
                       double m01,
                       double m11,
                       double m02,
                       double m12)
```

Constructs a new AffineTransform from 6 double precision values representing the 6 specifiable entries of the 3x3 transformation matrix.

o **AffineTransform**

```
public AffineTransform(double flatmatrix[])
```

Constructs a new AffineTransform from an array of double precision values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix. The values are retrieved from the array as { m00 m10 m01 m11 [m02 m12] }.

Parameters:
flatmatrix − The double array containing the values to be set in the new AffineTransform object. The length of the array is assumed to be at least 4. If the length of the array is less than 6, only the first 4 values are taken. If the length of the array is greater than 6, the first 6 values are taken.

o **AffineTransform**

```
public AffineTransform(double matrix[][])
```

Constructs a new AffineTransform from a 2−dimensional array of double precision values representing either the 4 non−translation or the 6 specifiable entries of the 3x3 transformation matrix. The values are retrieved from the 2−dimensional array using the row index as the first subscript and the column index as the second.

Parameters:
matrix − The matrix used to construct the new AffineTransform object. Matrix is assumed to be at least 2x2. If the matrix is smaller than 2x3, only the left upper 2x2 matrix is taken. If the matrix is larger than 2x3, the left upper 2x3 matrix is taken.

Methods

o **getType**

```
public int getType()
```

Retrieves the flag bits describing the conversion properties of this transform. The return value will be a combination of the appriopriate conversion flag bits.

Returns:
the OR combination of any of the indicated flags that apply to this transform
See Also:
TRANSLATION, UNIFORM_SCALE, GENERAL_SCALE, QUADRANT_ROTATION, GENERAL_ROTATION, GENERAL_TRANSFORM

o **getMatrix**

```
public void getMatrix(double flatmatrix[])
```

Retrieves the 6 specifiable values in the 3x3 affine transformation matrix into an array of double precisions values. The values are stored into the array as { m00 m10 m01 m11 m02 m12 }.

Parameters:

flatmatrix − The double array used to store the returned values. The length of the array is assumed to be at least 6.

See Also:

getScaleX, getScaleY, getShearX, getShearY, getTranslateX, getTranslateY

o **getMatrix**

```
public void getMatrix(double matrix[][])
```

Retrieves the 6 specifiable values in the 3x3 affine transformation matrix into a 2−dimensional array of double precision values. The values are stored into the 2−dimensional array using the row index as the first subscript and the column index as the second.

Parameters:

matrix − The 2−dimensional double array to store the returned values. The array is assumed to be at least 2x3.

See Also:

getScaleX, getScaleY, getShearX, getShearY, getTranslateX, getTranslateY

o **getScaleX**

```
public double getScaleX()
```

Returns the X coordinate scaling element (m00) of the 3x3 affine transformation matrix.

See Also:

getMatrix

o **getScaleY**

```
public double getScaleY()
```

Returns the Y coordinate scaling element (m11) of the 3x3 affine transformation matrix.

See Also:

getMatrix

o **getShearX**

```
public double getShearX()
```

Returns the X coordinate shearing element (m01) of the 3x3 affine transformation matrix.

See Also:

getMatrix

o **getShearY**

```
public double getShearY()
```

Returns the Y coordinate shearing element (m10) of the 3x3 affine transformation matrix.

See Also:
getMatrix

o **getTranslateX**

```
public double getTranslateX()
```

Returns the X coordinate translation element (m02) of the 3x3 affine transformation matrix.

See Also:
getMatrix

o **getTranslateY**

```
public double getTranslateY()
```

Returns the Y coordinate translation element (m12) of the 3x3 affine transformation matrix.

See Also:
getMatrix

o **setToIdentity**

```
public void setToIdentity()
```

Resets this transform to the Identity transform.

o **translate**

```
public void translate(double tx,
                      double ty)
```

Concatenates this transform with a translation transformation. This is equivalent to calling concatenate(T), where T is an AffineTransform represented by the following matrix:

```
[   1    0    tx  ]
[   0    1    ty  ]
[   0    0    1   ]
```

o **rotate**

```
public void rotate(double theta)
```

Concatenates this transform with a rotation transformation. This is equivalent to calling concatenate(R), where R is an AffineTransform represented by the following matrix:

```
[   cos(theta)    -sin(theta)    0   ]
[   sin(theta)     cos(theta)    0   ]
[       0              0         1   ]
```

Rotating with a positive angle theta rotates points on the positive x axis toward the positive y axis.

Parameters:

theta – The angle of rotation in radians.

o **rotate**

```
public void rotate(double theta,
                   double x,
                   double y)
```

Concatenates this transform with a translated rotation transformation. This is equivalent to the following sequence of calls:

```
translate(x, y);
rotate(theta);
translate(-x, -y);
```

Rotating with a positive angle theta rotates points on the positive x axis toward the positive y axis.

Parameters:

theta – The angle of rotation in radians.
x – The x coordinate of the origin of the rotation
y – The x coordinate of the origin of the rotation

o **scale**

```
public void scale(double sx,
                  double sy)
```

Concatenates this transform with a scaling transformation. This is equivalent to calling concatenate(S), where S is an AffineTransform represented by the following matrix:

```
[   sx   0    0   ]
[   0    sy   0   ]
[   0    0    1   ]
```

o **shear**

```
public void shear(double shx,
                  double shy)
```

Concatenates this transform with a shearing transformation. This is equivalent to calling concatenate(SH), where SH is an AffineTransform represented by the following matrix:

```
[   1   shx   0   ]
[  shy   1    0   ]
[   0    0    1   ]
```

Parameters:

shx – The factor by which coordinates are shifted towards the positive X axis direction according to their Y coordinate
shy – The factor by which coordinates are shifted towards the positive Y axis direction according to their X coordinate

o **setToTranslation**

```
public void setToTranslation(double tx,
                             double ty)
```

Sets this transform to a translation transformation. The matrix representing this transform becomes:

```
[   1    0    tx  ]
[   0    1    ty  ]
[   0    0    1   ]
```

o **setToRotation**

```
public void setToRotation(double theta)
```

Sets this transform to a rotation transformation. The matrix representing this transform becomes:

```
[   cos(theta)    -sin(theta)    0   ]
[   sin(theta)     cos(theta)    0   ]
[       0              0         1   ]
```

Rotating with a positive angle theta rotates points on the positive x axis toward the positive y axis.

Parameters:

theta – The angle of rotation in radians.

o **setToRotation**

```
public void setToRotation(double theta,
```

```
                          double x,
                          double y)
```

Sets this transform to a translated rotation transformation. This is equivalent to the following sequence of calls:

```
setToTranslation(x, y);
rotate(theta);
translate(-x, -y);
```

The matrix representing this transform becomes:

```
[   cos(theta)    -sin(theta)    x-x*cos+y*sin  ]
[   sin(theta)     cos(theta)    y-x*sin-y*cos  ]
[       0              0               1        ]
```

Rotating with a positive angle theta rotates points on the positive x axis toward the positive y axis.

Parameters:

theta – The angle of rotation in radians.
x – The x coordinate of the origin of the rotation
y – The x coordinate of the origin of the rotation

o **setToScale**

```
public void setToScale(double sx,
                       double sy)
```

Sets this transform to a scaling transformation. The matrix representing this transform becomes:

```
[   sx    0     0   ]
[   0     sy    0   ]
[   0     0     1   ]
```

o **setToShear**

```
public void setToShear(double shx,
                       double shy)
```

Sets this transform to a shearing transformation. The matrix representing this transform becomes:

```
[   1    shx    0   ]
[  shy    1     0   ]
[   0     0     1   ]
```

Parameters:

shx – The factor by which coordinates are shifted towards the positive X axis direction according to their Y coordinate
shy – The factor by which coordinates are shifted towards the positive Y axis direction according to their X coordinate

o **concatenate**

```
public void concatenate(AffineTransform Tx)
```

Concatenates an AffineTransform Tx to this AffineTransform Cx in the most commonly useful way to provide a new user space which is mapped to the former user space by Tx. Cx is updated to perform the combined transformation. Transforming a point p by the updated transform Cx' is equivalent to first transforming p by Tx and then transforming the result by the original transform Cx. In other words, Cx'(p) = Cx(Tx(p)). In matrix notation, if this transform Cx is represented by the matrix [this] and Tx is represented by the matrix [Tx], then this method does the following:

```
[this] = [this] x [Tx]
```

Parameters:
Tx – The AffineTransform object to be concatenated with this AffineTransform object.
See Also:
preConcatenate

o **prepend**

```
public void prepend(AffineTransform Tx)
```

Note: prepend() is deprecated. *As of JDK version 1.2beta, replaced by* `concatenate(AffineTransform)`. *This method will disappear in the next release.*

o **preConcatenate**

```
public void preConcatenate(AffineTransform Tx)
```

Concatenates an AffineTransform Tx to this AffineTransform Cx in a less commonly used way such that Tx modifies the coordinate transformation relative to the absolute pixel space rather than relative to the existing user space. Cx is updated to perform the combined transformation. Transforming a point p by the updated transform Cx' is equivalent to first transforming p by the original transform Cx and then transforming the result by Tx. In other words, Cx'(p) = Tx(Cx(p)). In matrix notation, if this transform Cx is represented by the matrix [this] and Tx is represented by the matrix [Tx], then this method does the following:

```
[this] = [Tx] x [this]
```

Parameters:
Tx – The AffineTransform object to be concatenated with this AffineTransform object.

See Also:
concatenate

o **append**

```
public void append(AffineTransform Tx)
```

Note: append() is deprecated. *As of JDK version 1.2beta, replaced by* `preConcatenate(AffineTransform)`. *This method will disappear in the next release.*

o **createInverse**

```
public AffineTransform createInverse() throws NoninvertibleTransformException
```

Returns a Transform object representing the inverse transformation.

Throws: NoninvertibleTransformException
if the matrix cannot be inverted.

o **transform**

```
public Point2D transform(Point2D ptSrc,
                         Point2D ptDst)
```

Transforms the specified ptSrc and stores the result in ptDst. If ptDst is null, a new Point2D object will be allocated before storing. In either case, ptDst containing the transformed point is returned for convenience. Note that ptSrc and ptDst can the same. In this case, the input point will be overwritten with the transformed point.

o **transform**

```
public void transform(Point2D ptSrc[],
                      int srcOff,
                      Point2D ptDst[],
                      int dstOff,
                      int numPts)
```

Transforms an array of point objects by this transform.

Parameters:
ptSrc – The array containing the source point objects.
ptDst – The array where the transform point objects are returned.
srcOff – The offset to the first point object to be transformed in the source array.
dstOff – The offset to the location where the first transformed point object is stored in the destination array.
numPts – The number of point objects to be transformed.

o **transform**

```
public void transform(float srcPts[],
                      int srcOff,
                      float dstPts[],
                      int dstOff,
                      int numPts)
```

Transforms an array of floating point coordinates by this transform.

Parameters:

ptSrc – The array containing the source point coordinates. Each point is stored as a pair of x,y coordinates.
ptDst – The array where the transformed point coordinates are returned. Each point is stored as a pair of x,y coordinates.
srcOff – The offset to the first point to be transformed in the source array.
dstOff – The offset to the location where the first transformed point is stored in the destination array.
numPts – The number of points to be transformed.

o **transform**

```
public void transform(double srcPts[],
                      int srcOff,
                      double dstPts[],
                      int dstOff,
                      int numPts)
```

Transforms an array of double precision coordinates by this transform.

Parameters:

srcPts – The array containing the source point coordinates. Each point is stored as a pair of x,y coordinates.
dstPts – The array where the transformed point coordinates are returned. Each point is stored as a pair of x,y coordinates.
srcOff – The offset to the first point to be transformed in the source array.
dstOff – The offset to the location where the first transformed point is stored in the destination array.
numPts – The number of point objects to be transformed.

o **transform**

```
public void transform(float srcPts[],
                      int srcOff,
                      double dstPts[],
                      int dstOff,
                      int numPts)
```

Transforms an array of floating point coordinates by this transform, storing the results into an array of doubles.

Parameters:

ptSrc – The array containing the source point coordinates. Each point is stored as a pair

of x,y coordinates.
ptDst – The array where the transformed point coordinates are returned. Each point is stored as a pair of x,y coordinates.
srcOff – The offset to the first point to be transformed in the source array.
dstOff – The offset to the location where the first transformed point is stored in the destination array.
numPts – The number of points to be transformed.

o **transform**

```
public void transform(double srcPts[],
                      int srcOff,
                      float dstPts[],
                      int dstOff,
                      int numPts)
```

Transforms an array of double precision coordinates by this transform, storing the results into an array of floats.

Parameters:
srcPts – The array containing the source point coordinates. Each point is stored as a pair of x,y coordinates.
dstPts – The array where the transformed point coordinates are returned. Each point is stored as a pair of x,y coordinates.
srcOff – The offset to the first point to be transformed in the source array.
dstOff – The offset to the location where the first transformed point is stored in the destination array.
numPts – The number of point objects to be transformed.

o **inverseTransform**

```
public Point2D inverseTransform(Point2D ptSrc,
                                Point2D ptDst) throws NoninvertibleTransformException
```

Inverse transforms the specified ptSrc and stores the result in ptDst. If ptDst is null, a new Point2D object will be allocated before storing. In either case, ptDst containing the transformed point is returned for convenience. Note that ptSrc and ptDst can the same. In this case, the input point will be overwritten with the transformed point.

Parameters:
ptSrc – The point to be inverse transformed.
ptDst – The resulting transformed point.
Throws: NoninvertibleTransformException
if the matrix cannot be inverted.

o **inverseTransform**

```
public void inverseTransform(double srcPts[],
                             int srcOff,
                             double dstPts[],
```

```
                                      int dstOff,
                                      int numPts) throws NoninvertibleTransformException
```

Inverse transforms an array of double precision coordinates by this transform.

Parameters:

srcPts – The array containing the source point coordinates. Each point is stored as a pair of x,y coordinates.
dstPts – The array where the transformed point coordinates are returned. Each point is stored as a pair of x,y coordinates.
srcOff – The offset to the first point to be transformed in the source array.
dstOff – The offset to the location where the first transformed point is stored in the destination array.
numPts – The number of point objects to be transformed.

Throws: NoninvertibleTransformException

if the matrix cannot be inverted.

o **createTransformedShape**

```
public Shape createTransformedShape(Shape pSrc)
```

Transforms a shape object by this transform.

o **toString**

```
public String toString()
```

Returns a String that represents the value of this Object.

Overrides:

toString in class Object

o **isIdentity**

```
public boolean isIdentity()
```

Returns the boolean true value if this AffineTransform is an identity transform. Returns false otherwise.

o **clone**

```
public Object clone()
```

Returns a copy of this AffineTransform object.

Overrides:

clone in class Object

o **equals**

```
public boolean equals(Object obj)
```

Overrides:
equals in class Object

Class java.awt.geom.Arc2D

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Arc2D
```

public abstract class **Arc2D**
extends RectangularShape
super class of:
    Arc2D.Float

An arc defined by a bounding rectangle, start angle and angular extent, and a closure type (one of OPEN, CHORD, or PIE). The bounding rectangle defines the outer boundary of the full ellipse of which this arc is a partial section. The angles are specified relative to the non−square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the bounds.

This class is only the abstract superclass for all objects which store a 2D arc. The actual storage representation of the coordinates is left to the subclass.

Class Index

o **Arc2D.Float**
    An arc specified in float precision,

Variable Index

o **CHORD**
    The closure type for an arc closed by drawing a straight line segment from the start to the end of the arc segment.
o **OPEN**
    The closure type for an open arc with no path segments connecting the two ends of the arc segment.

o **PIE**
The closure type for an arc closed by drawing straight line segments from the start of the arc segment to the center of the full ellipse and back to the end of the arc segment.

Constructor Index

o **Arc2D**(int)
Constructs a new arc of the specified closure type.

Method Index

o **contains**(double, double)
Test if a given Point is inside the boundary of the shape.
o **contains**(double, double, double, double)
Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
o **contains**(Rectangle2D)
Test if the interior of the Shape entirely contains the given Rectangle.
o **containsAngle**(double)
Tests if a given angle is within the angular extents of the arc.
o **getAngleExtent**()
Returns the arc length (angular extent) of the arc (in degrees).
o **getAngleStart**()
Returns the starting angle of the arc (in degrees).
o **getArcType**()
Returns the arc closure type of the arc.
o **getBounds2D**()
Return the high precision bounding box of the shape.
o **getEndPoint**()
Returns the ending point of the arc.
o **getPathIterator**(AffineTransform)
Return an iteration object that defines the boundary of the shape.
o **getStartPoint**()
Returns the starting point of the arc.
o **intersects**(double, double, double, double)
Test if the interior of the Shape intersects the interior of a given Rectangle.
o **makeBounds**(double, double, double, double)
Construct a rectangle object of the appropriate precision to hold the parameters calculated to be the bounding box of this arc.
o **normalizeDegrees**(double)
o **setAngleExtent**(double)
Sets the angular extent of this arc to the specified double value (in degrees).
o **setAngles**(double, double, double, double)
Sets the angular extents of this arc to the angles that the specified point coordinates define relative to the center of this arc.
o **setAngles**(Point2D, Point2D)
Sets the angular extents of this arc to the angles that the specified Point objects define relative to the center of this arc.

o **setAngleStart**(double)
  Sets the starting angle of this arc to the specified double value (in degrees).
o **setAngleStart**(Point2D)
  Sets the starting angle of this arc to the angle that the specified point defines relative to the center of this arc.
o **setArc**(Arc2D)
  Sets this arc to be the same as the specified Arc.
o **setArc**(double, double, double, double, double, double, int)
  Sets the location, size, angular extents, and closure type of this arc to the specified double values.
o **setArc**(Point2D, Dimension2D, double, double, int)
  Sets the location, size, angular extents, and closure type of this arc to the specified Point and Dimension and double values.
o **setArc**(Rectangle2D, double, double, int)
  Sets the location, size, angular extents, and closure type of this arc to the specified Rectangle and double values.
o **setArcByCenter**(double, double, double, double, double, int)
  Sets the position, bounds, angular extents, and closure type of this arc to the specified double values based on a center point and a radius rather than a bounding box for the full ellipse.
o **setArcByTangent**(Point2D, Point2D, Point2D, double)
  Sets the position, bounds, and angular extents of this arc so that the starting angle is tangent to the line specified by points (p1, p2) and the ending angle is tangent to the line specified by points (p2, p3) with the given radius.
o **setArcType**(int)
  Sets the closure type of this arc to the specified value.
o **setBounds**(double, double, double, double)
  Sets the location and size of the outer bounds of this shape to the specified rectangular values.
o **toDegrees**(double)
o **toRadians**(double)

Variables

o **OPEN**

```
public static final int OPEN
```

The closure type for an open arc with no path segments connecting the two ends of the arc segment.

o **CHORD**

```
public static final int CHORD
```

The closure type for an arc closed by drawing a straight line segment from the start to the end of the arc segment.

o **PIE**

```
public static final int PIE
```

The closure type for an arc closed by drawing straight line segments from the start of the arc segment to the center of the full ellipse and back to the end of the arc segment.

Constructors

o **Arc2D**

```
protected Arc2D(int type)
```

Constructs a new arc of the specified closure type.

Methods

o **getAngleStart**

```
public abstract double getAngleStart()
```

Returns the starting angle of the arc (in degrees).

o **getAngleExtent**

```
public abstract double getAngleExtent()
```

Returns the arc length (angular extent) of the arc (in degrees).

o **getArcType**

```
public int getArcType()
```

Returns the arc closure type of the arc.

See Also:
OPEN, CHORD, PIE

o **getStartPoint**

```
public Point2D getStartPoint()
```

Returns the starting point of the arc. This point is the intersection of the ray from the center defined by the starting angle and the elliptical boundary of the arc.

o **getEndPoint**

```
public Point2D getEndPoint()
```

Returns the ending point of the arc. This point is the intersection of the ray from the center defined by the starting angle plus the angular extent of the arc and the elliptical boundary of the arc.

o **setArc**

```
public abstract void setArc(double x,
                            double y,
                            double w,
                            double h,
                            double angSt,
                            double angExt,
                            int closure)
```

Sets the location, size, angular extents, and closure type of this arc to the specified double values.

o **setArc**

```
public void setArc(Point2D loc,
                   Dimension2D size,
                   double angSt,
                   double angExt,
                   int closure)
```

Sets the location, size, angular extents, and closure type of this arc to the specified Point and Dimension and double values.

o **setArc**

```
public void setArc(Rectangle2D rect,
                   double angSt,
                   double angExt,
                   int closure)
```

Sets the location, size, angular extents, and closure type of this arc to the specified Rectangle and double values.

o **setArc**

```
public void setArc(Arc2D a)
```

Sets this arc to be the same as the specified Arc.

o **setArcByCenter**

```
public void setArcByCenter(double x,
                           double y,
                           double radius,
                           double angSt,
                           double angExt,
                           int closure)
```

Sets the position, bounds, angular extents, and closure type of this arc to the specified double values based on a center point and a radius rather than a bounding box for the full ellipse.

o **setArcByTangent**

```
public void setArcByTangent(Point2D p1,
                            Point2D p2,
                            Point2D p3,
                            double radius)
```

Sets the position, bounds, and angular extents of this arc so that the starting angle is tangent to the line specified by points (p1, p2) and the ending angle is tangent to the line specified by points (p2, p3) with the given radius.

o **setAngleStart**

```
public abstract void setAngleStart(double angSt)
```

Sets the starting angle of this arc to the specified double value (in degrees).

o **setAngleExtent**

```
public abstract void setAngleExtent(double angExt)
```

Sets the angular extent of this arc to the specified double value (in degrees).

o **setAngleStart**

```
public void setAngleStart(Point2D p)
```

Sets the starting angle of this arc to the angle that the specified point defines relative to the center of this arc. The angular extent of the arc will remain the same.

o **setAngles**

```
public void setAngles(double x1,
                      double y1,
                      double x2,
                      double y2)
```

Sets the angular extents of this arc to the angles that the specified point coordinates define relative to the center of this arc. The arc will always be non−empty and extend counterclockwise from the first point around to the second point.

o **setAngles**

```
public void setAngles(Point2D p1,
                      Point2D p2)
```

Sets the angular extents of this arc to the angles that the specified Point objects define relative to the center of this arc. The arc will always be non−empty and extend counterclockwise from the first point around to the second point.

o **setArcType**

```
public void setArcType(int type)
```

Sets the closure type of this arc to the specified value.

See Also:
OPEN, CHORD, PIE

o **setBounds**

```
public void setBounds(double x,
                      double y,
                      double w,
                      double h)
```

Sets the location and size of the outer bounds of this shape to the specified rectangular values.

Overrides:
setBounds in class RectangularShape

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

o **makeBounds**

```
protected abstract Rectangle2D makeBounds(double x,
                                          double y,
                                          double w,
                                          double h)
```

Construct a rectangle object of the appropriate precision to hold the parameters calculated to be the bounding box of this arc.

o **toDegrees**

```
protected static double toDegrees(double angrad)
```

o **toRadians**

```
protected static double toRadians(double angdeg)
```

o **normalizeDegrees**

```
protected static double normalizeDegrees(double angle)
```

o **containsAngle**

```
public boolean containsAngle(double angle)
```

Tests if a given angle is within the angular extents of the arc.

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the interior of the Shape intersects the interior of a given Rectangle.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

Overrides:
contains in class RectangularShape

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

Class java.awt.geom.Arc2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Arc2D
                   |
                   +----java.awt.geom.Arc2D.Float
```

---

public static class **Arc2D.Float**
extends Arc2D

An arc specified in float precision,

Variable Index

- o **extent**
  The angular extent of the arc in degrees.
- o **height**
  The overall height of the full ellipse (not considering the angular extents).
- o **start**
  The starting angle of the arc in degrees.
- o **width**
  The overall width of the full ellipse (not considering the angular extents).
- o **x**
  The x coordinate of the upper left corner of the arc.
- o **y**
  The y coordinate of the upper left corner of the arc.

Constructor Index

- o **Arc2D.Float**()
  Constructs a new OPEN arc, initialized to location (0, 0), size (0, 0), angular extents (start = 0, extent = 0).
- o **Arc2D.Float**(float, float, float, float, float, float, int)
  Constructs a new arc, initialized to the specified location, size, angular extents, and closure

type.

o **Arc2D.Float**(int)
: Constructs a new arc, initialized to location (0, 0), size (0, 0), angular extents (start = 0, extent = 0), and the specified closure type.

o **Arc2D.Float**(Rectangle2D, float, float, int)
: Constructs a new arc, initialized to the specified location, size, angular extents, and closure type.

Method Index

o **getAngleExtent**()
: Returns the arc length (angular extent) of the arc (in degrees).

o **getAngleStart**()
: Returns the starting angle of the arc (in degrees).

o **getHeight**()
: Returns the overall height of the arc in double precision.

o **getWidth**()
: Returns the overall width of the arc in double precision.

o **getX**()
: Returns the X coordinate of the upper left corner of the arc in double precision.

o **getY**()
: Returns the Y coordinate of the upper left corner of the arc in double precision.

o **isEmpty**()
: Determines whether the rectangular shape is empty.

o **makeBounds**(double, double, double, double)
: Return the high precision bounding box of the shape.

o **setAngleExtent**(double)
: Sets the angular extent of this arc to the specified double value (in degrees).

o **setAngleStart**(double)
: Sets the starting angle of this arc to the specified double value (in degrees).

o **setArc**(double, double, double, double, double, double, int)
: Sets the location, size, angular extents, and closure type of this arc to the specified double values.

Variables

o **x**

```
public float x
```

The x coordinate of the upper left corner of the arc.

o **y**

```
public float y
```

The y coordinate of the upper left corner of the arc.

o **width**

```
public float width
```

The overall width of the full ellipse (not considering the angular extents).

o **height**

```
public float height
```

The overall height of the full ellipse (not considering the angular extents).

o **start**

```
public float start
```

The starting angle of the arc in degrees.

o **extent**

```
public float extent
```

The angular extent of the arc in degrees.

Constructors

o **Arc2D.Float**

```
public Arc2D.Float()
```

Constructs a new OPEN arc, initialized to location (0, 0), size (0, 0), angular extents (start = 0, extent = 0).

o **Arc2D.Float**

```
public Arc2D.Float(int type)
```

Constructs a new arc, initialized to location (0, 0), size (0, 0), angular extents (start = 0, extent = 0), and the specified closure type.

o **Arc2D.Float**

```
public Arc2D.Float(float x,
                   float y,
                   float w,
                   float h,
                   float start,
                   float extent,
                   int type)
```

Constructs a new arc, initialized to the specified location, size, angular extents, and closure type.

o **Arc2D.Float**

```
public Arc2D.Float(Rectangle2D ellipseBounds,
                   float start,
                   float extent,
                   int type)
```

Constructs a new arc, initialized to the specified location, size, angular extents, and closure type.

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the upper left corner of the arc in double precision.

Overrides:
getX in class RectangularShape

o **getY**

```
public double getY()
```

Returns the Y coordinate of the upper left corner of the arc in double precision.

Overrides:
getY in class RectangularShape

o **getWidth**

```
public double getWidth()
```

Returns the overall width of the arc in double precision.

Overrides:
getWidth in class RectangularShape

o **getHeight**

```
public double getHeight()
```

Returns the overall height of the arc in double precision.

Overrides:
getHeight in class RectangularShape

o **getAngleStart**

```
public double getAngleStart()
```

Returns the starting angle of the arc (in degrees).

Overrides:
getAngleStart in class Arc2D

o **getAngleExtent**

```
public double getAngleExtent()
```

Returns the arc length (angular extent) of the arc (in degrees).

Overrides:
getAngleExtent in class Arc2D

o **isEmpty**

```
public boolean isEmpty()
```

Determines whether the rectangular shape is empty.

Overrides:
isEmpty in class RectangularShape

o **setArc**

```
public void setArc(double x,
                   double y,
                   double w,
                   double h,
                   double angSt,
                   double angExt,
                   int closure)
```

Sets the location, size, angular extents, and closure type of this arc to the specified double values.

Overrides:
setArc in class Arc2D

o **setAngleStart**

```
public void setAngleStart(double angSt)
```

Sets the starting angle of this arc to the specified double value (in degrees).

Overrides:
setAngleStart in class Arc2D

o **setAngleExtent**

```
public void setAngleExtent(double angExt)
```

Sets the angular extent of this arc to the specified double value (in degrees).

Overrides:
setAngleExtent in class Arc2D

o **makeBounds**

```
public Rectangle2D makeBounds(double x,
                              double y,
                              double w,
                              double h)
```

Return the high precision bounding box of the shape.

Overrides:
makeBounds in class Arc2D

Class java.awt.geom.Area

```
java.lang.Object
   |
   +----sun.awt.Albert.Area
           |
           +----java.awt.geom.Area
```

public class **Area**
extends Area

Variable Index

o **EmptyArea**
  Constant area geometry that is empty.

Constructor Index

o **Area**()
  Default constructor which creates an empty area.
o **Area**(Ellipse2D)
  Creates an area geometry of a ellipse inside the specified rectangle.
o **Area**(Polygon)
  Creates an area geometry from the specified polygon.
o **Area**(Rectangle2D)
  Creates an area geometry from the specified rectangle.
o **Area**(Shape)
  Creates an area geometry from the specified Shape object.

Method Index

o **add**(Area)
  Adds the shape of the specified Area to the current shape.
o **contains**(double, double)
  Test if a given coordinate is inside the boundary of the shape.
o **contains**(double, double, double, double)

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **contains**(Point2D)
: Test if a given Point is inside the boundary of the shape.

o **contains**(Rectangle2D)
: Test if the interior of the Shape entirely contains the given Rectangle.

o **exclusiveOr**(Area)
: Sets the shape to the combined area of the current shape and the shape of the specified Area, minus their intersection.

o **getBounds**()
: Return the bounding box of the shape.

o **getBounds2D**()
: Return the high precision bounding box of the shape.

o **getPathIterator**(AffineTransform)
: Return an iterator object that iterates along the boundary of the shape and provides access to the geometry of the outline of the shape.

o **getPathIterator**(AffineTransform, double)
: Return an iterator object that iterates along the boundary of the shape and provides access to a flattened view of the geometry of the outline of the shape.

o **intersect**(Area)
: Sets the shape to the intersection of the current shape with the shape of the specified Area.

o **intersects**(double, double, double, double)
: Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.

o **intersects**(Rectangle2D)
: Test if the interior of the Shape intersects the interior of a given Rectangle.

o **isEmpty**()
: Tests whether this area contains any geometries.

o **isPolygonal**()
: Tests whether the area comprises completely of horizontal and vertical edges.

o **isRectangular**()
: Tests whether the area is rectangular in shape.

o **isSingular**()
: Tests whether the area is comprised of a single basic geometry.

o **reset**()
: Removes all the basic geometries from this area and results in an empty area.

o **subtract**(Area)
: Subtracts the shape of the specified Area from the current shape.

Variables

o **EmptyArea**

```
public static Area EmptyArea
```

Constant area geometry that is empty. Rendering such geometry will have no effect (NOP).

Constructors

o **Area**

```
public Area()
```

Default constructor which creates an empty area.

o **Area**

```
public Area(Shape g)
```

Creates an area geometry from the specified Shape object. The shape is explicitly closed, if it is not closed. The fill option (even−odd or winding) specified by the shape geometry is used to determined the resulting area.

Parameters:
g − The shape from which the area is to be constructed.

o **Area**

```
public Area(Polygon p)
```

Creates an area geometry from the specified polygon. Even−odd fill option is assumed (because that is how awt Polygon is defined).

Parameters:
p − The polygon from which the area is to be constructed.

o **Area**

```
public Area(Rectangle2D r)
```

Creates an area geometry from the specified rectangle.

Parameters:
r − The rectangle from which the area is to be constructed.

o **Area**

```
public Area(Ellipse2D e)
```

Creates an area geometry of a ellipse inside the specified rectangle.

Parameters:
e − The ellipse geometry from which the area is to be constructed

Methods

o **getBounds**

```
public Rectangle getBounds()
```

Return the bounding box of the shape.

Overrides:
getBounds in class Area

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

Overrides:
getBounds2D in class Area

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

Overrides:
contains in class Area

o **contains**

```
public boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

Overrides:
contains in class Area

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.

Overrides:

intersects in class Area

o **intersects**

```
public boolean intersects(Rectangle2D r)
```

Test if the interior of the Shape intersects the interior of a given Rectangle.

Overrides:
intersects in class Area

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

Overrides:
contains in class Area

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

Overrides:
contains in class Area

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iterator object that iterates along the boundary of the shape and provides access to the geometry of the outline of the shape. An optional affine transform can be specified in which case the coordinates returned in the iteration will be transformed accordingly.

Parameters:
at – an optional AffineTransform to be applied to the coordinates as they are returned in the iteration, or null if the untransformed coordinates are desired.

Overrides:
getPathIterator in class Area

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
```

```
double flatness)
```

Return an iterator object that iterates along the boundary of the shape and provides access to a flattened view of the geometry of the outline of the shape. Only SEG_MOVETO, SEG_LINETO, and SEG_CLOSE point types will be returned by the iterator. The amount of subdivision of the curved segments is controlled by the `flatness` parameter which specifies ?REMIND?. An optional affine transform can be specified in which case the coordinates returned in the iteration will be transformed accordingly.

Parameters:
at – an optional AffineTransform to be applied to the coordinates as they are returned in the iteration, or null if the untransformed coordinates are desired.
flatness – the maximum amount that the control points for a given curve can vary from colinear before a subdivided curve is replaced by a straight line connecting the endpoints.

Overrides:
getPathIterator in class Area

o **reset**

```
public void reset()
```

Removes all the basic geometries from this area and results in an empty area.

Overrides:
reset in class Area

o **isEmpty**

```
public boolean isEmpty()
```

Tests whether this area contains any geometries.

Returns:
True if this area contains no basic geometries, or is an empty area.

Overrides:
isEmpty in class Area

o **isPolygonal**

```
public boolean isPolygonal()
```

Tests whether the area comprises completely of horizontal and vertical edges.

Returns:
True if the area comprises completely of horizontal and vertical edges.

Overrides:
isPolygonal in class Area

o **isRectangular**

```
public boolean isRectangular()
```

Tests whether the area is rectangular in shape.

Returns:
True if the area is rectangular in shape.
Overrides:
isRectangular in class Area

o **isSingular**

```
public boolean isSingular()
```

Tests whether the area is comprised of a single basic geometry.

Returns:
True if the area is comprised of a single basic geometry.
Overrides:
isSingular in class Area

o **add**

```
public void add(Area rhs)
```

Adds the shape of the specified Area to the current shape. Addition is achieved through union.

Parameters:
rhs – The shape to be added to the current area.

o **subtract**

```
public void subtract(Area rhs)
```

Subtracts the shape of the specified Area from the current shape.

Parameters:
rhs – The shape to be subtracted from the current area.

o **intersect**

```
public void intersect(Area rhs)
```

Sets the shape to the intersection of the current shape with the shape of the specified Area.

Parameters:
rhs – The area to be intersected with this one.

o **exclusiveOr**

```
public void exclusiveOr(Area rhs)
```

Sets the shape to the combined area of the current shape and the shape of the specified Area, minus their intersection.

Parameters:

rhs – The area to be exclusive ORed with this one.

Class java.awt.geom.CubicCurve2D

```
java.lang.Object
   |
   +----java.awt.geom.CubicCurve2D
```

---

public abstract class **CubicCurve2D**
extends Object
implements Shape
super class of:
CubicCurve2D.Float

A cubic parametric curve segment in (x, y) coordinate space.

This class is only the abstract superclass for all objects which store a 2D cubic curve segment. The actual storage representation of the coordinates is left to the subclass.

Class Index

- **CubicCurve2D.Float**
  A cubic parametric curve segment specified with float coordinates.

Constructor Index

- **CubicCurve2D**()

Method Index

- **contains**(double, double)
  Test if a given coordinate is inside the boundary of the shape.
- **contains**(double, double, double, double)
  Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
- **contains**(Point2D)
  Test if a given Point is inside the boundary of the shape.
- **contains**(Rectangle2D)
  Test if the interior of the Shape entirely contains the given Rectangle.

- o **getBounds**()
  Return the bounding box of the shape.
- o **getCtrlX1**()
  Returns the X coordinate of the first control point in double precision.
- o **getCtrlX2**()
  Returns the X coordinate of the second control point in double precision.
- o **getCtrlY1**()
  Returns the Y coordinate of the first control point in double precision.
- o **getCtrlY2**()
  Returns the Y coordinate of the second control point in double precision.
- o **getFlatness**()
  Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.
- o **getFlatness**(double, double, double, double, double, double, double, double)
  Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the indicated controlpoints.
- o **getFlatness**(double[], int)
  Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the controlpoints stored in the indicated array at the indicated index.
- o **getFlatnessSq**()
  Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.
- o **getFlatnessSq**(double, double, double, double, double, double, double, double)
  Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the indicated controlpoints.
- o **getFlatnessSq**(double[], int)
  Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the controlpoints stored in the indicated array at the indicated index.
- o **getPathIterator**(AffineTransform)
  Return an iteration object that defines the boundary of the shape.
- o **getPathIterator**(AffineTransform, double)
  Return an iteration object that defines the boundary of the flattened shape.
- o **getX1**()
  Returns the X coordinate of the start point in double precision.
- o **getX2**()
  Returns the X coordinate of the end point in double precision.
- o **getY1**()
  Returns the Y coordinate of the start point in double precision.
- o **getY2**()
  Returns the Y coordinate of the end point in double precision.
- o **intersects**(double, double, double, double)
  Test if the Shape intersects the interior of a given set of rectangular coordinates.
- o **intersects**(Rectangle2D)
  Test if the Shape intersects the interior of a given Rectangle.
- o **setCurve**(CubicCurve2D)
  Sets the location of the endpoints and controlpoints of this curve to the same as those in the specified CubicCurve.

o **setCurve**(double, double, double, double, double, double, double, double)
Sets the location of the endpoints and controlpoints of this curve to the specified double coordinates.

o **setCurve**(double[], int)
Sets the location of the endpoints and controlpoints of this curve to the double coordinates at the specified offset in the specified array.

o **setCurve**(Point2D, Point2D, Point2D, Point2D)
Sets the location of the endpoints and controlpoints of this curve to the specified Point coordinates.

o **setCurve**(Point2D[], int)
Sets the location of the endpoints and controlpoints of this curve to the coordinates of the Point objects at the specified offset in the specified array.

o **solveCubic**(double[])
Solve the cubic whose coefficients are in the eqn array and place the non−complex roots back into the array, returning the number of roots.

o **subdivide**(CubicCurve2D, CubicCurve2D)
Subdivides this cubic curve and stores the resulting two subdivided curves into the left and right curve parameters.

o **subdivide**(CubicCurve2D, CubicCurve2D, CubicCurve2D)
Subdivides the cubic curve specified by the src parameter and stores the resulting two subdivided curves into the left and right curve parameters.

o **subdivide**(double[], int, double[], int, double[], int)
Subdivides the cubic curve specified by the the coordinates stored in the src array at indices (srcoff) through (srcoff + 7) and stores the resulting two subdivided curves into the two result arrays at the corresponding indices.

Constructors

o **CubicCurve2D**

```
protected CubicCurve2D()
```

Methods

o **getX1**

```
public abstract double getX1()
```

Returns the X coordinate of the start point in double precision.

o **getY1**

```
public abstract double getY1()
```

Returns the Y coordinate of the start point in double precision.

o **getCtrlX1**

```
public abstract double getCtrlX1()
```

Returns the X coordinate of the first control point in double precision.

o **getCtrlY1**

```
public abstract double getCtrlY1()
```

Returns the Y coordinate of the first control point in double precision.

o **getCtrlX2**

```
public abstract double getCtrlX2()
```

Returns the X coordinate of the second control point in double precision.

o **getCtrlY2**

```
public abstract double getCtrlY2()
```

Returns the Y coordinate of the second control point in double precision.

o **getX2**

```
public abstract double getX2()
```

Returns the X coordinate of the end point in double precision.

o **getY2**

```
public abstract double getY2()
```

Returns the Y coordinate of the end point in double precision.

o **setCurve**

```
public abstract void setCurve(double x1,
                              double y1,
                              double ctrlx1,
                              double ctrly1,
                              double ctrlx2,
                              double ctrly2,
                              double x2,
                              double y2)
```

Sets the location of the endpoints and controlpoints of this curve to the specified double coordinates.

o **setCurve**

```
public void setCurve(double coords[],
                     int offset)
```

Sets the location of the endpoints and controlpoints of this curve to the double coordinates at the specified offset in the specified array.

o **setCurve**

```
public void setCurve(Point2D p1,
                     Point2D cp1,
                     Point2D cp2,
                     Point2D p2)
```

Sets the location of the endpoints and controlpoints of this curve to the specified Point coordinates.

o **setCurve**

```
public void setCurve(Point2D pts[],
                     int offset)
```

Sets the location of the endpoints and controlpoints of this curve to the coordinates of the Point objects at the specified offset in the specified array.

o **setCurve**

```
public void setCurve(CubicCurve2D c)
```

Sets the location of the endpoints and controlpoints of this curve to the same as those in the specified CubicCurve.

o **getFlatnessSq**

```
public static double getFlatnessSq(double x1,
                                   double y1,
                                   double ctrlx1,
                                   double ctrly1,
                                   double ctrlx2,
                                   double ctrly2,
                                   double x2,
                                   double y2)
```

Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the indicated controlpoints.

o **getFlatness**

```
public static double getFlatness(double x1,
                                 double y1,
                                 double ctrlx1,
                                 double ctrly1,
                                 double ctrlx2,
```

```
                                    double ctrly2,
                                    double x2,
                                    double y2)
```

Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the indicated controlpoints.

o **getFlatnessSq**

```
public static double getFlatnessSq(double coords[],
                                   int offset)
```

Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the controlpoints stored in the indicated array at the indicated index.

o **getFlatness**

```
public static double getFlatness(double coords[],
                                 int offset)
```

Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the cubic curve specified by the controlpoints stored in the indicated array at the indicated index.

o **getFlatnessSq**

```
public double getFlatnessSq()
```

Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.

o **getFlatness**

```
public double getFlatness()
```

Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.

o **subdivide**

```
public void subdivide(CubicCurve2D left,
                      CubicCurve2D right)
```

Subdivides this cubic curve and stores the resulting two subdivided curves into the left and right curve parameters. Either or both of the left and right objects may be the same as this object or null.

Parameters:

left – the cubic curve object for storing for the left or first half of the subdivided curve right − the cubic curve object for storing for the right or second half of the subdivided curve

o **subdivide**

```
public static void subdivide(CubicCurve2D src,
                             CubicCurve2D left,
                             CubicCurve2D right)
```

Subdivides the cubic curve specified by the src parameter and stores the resulting two subdivided curves into the left and right curve parameters. Either or both of the left and right objects may be the same as the src object or null.

Parameters:

src − the cubic curve to be subdivided
left − the cubic curve object for storing for the left or first half of the subdivided curve
right − the cubic curve object for storing for the right or second half of the subdivided curve

o **subdivide**

```
public static void subdivide(double src[],
                             int srcoff,
                             double left[],
                             int leftoff,
                             double right[],
                             int rightoff)
```

Subdivides the cubic curve specified by the the coordinates stored in the src array at indices (srcoff) through (srcoff + 7) and stores the resulting two subdivided curves into the two result arrays at the corresponding indices. Either or both of the left and right arrays may be null or a reference to the same array as the src array. Note that the last point in the first subdivided curve is the same as the first point in the second subdivided curve and thus it is possible to pass the same array for left and right and to use offsets such that rightoff equals (leftoff + 6) in order to avoid allocating extra storage for this common point.

Parameters:

src − the array holding the coordinates for the source curve
srcoff − the offset into the array of the beginning of the the 6 source coordinates
left − the array for storing the coordinates for the first half of the subdivided curve
leftoff − the offset into the array of the beginning of the the 6 left coordinates
right − the array for storing the coordinates for the second half of the subdivided curve
rightoff − the offset into the array of the beginning of the the 6 right coordinates

o **solveCubic**

```
public static int solveCubic(double eqn[])
```

Solve the cubic whose coefficients are in the eqn array and place the non−complex roots back into the array, returning the number of roots. The quadratic solved is represented by the equation: eqn = {c, b, a, d} dx^3 + ax^2 + bx + c = 0 A return value of −1 is used to distinguish a constant equation, which may be always 0 or never 0, from an equation which has no zeroes.

Returns:
the number of roots, or −1 if the equation is a constant

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **contains**

```
public boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the Shape intersects the interior of a given set of rectangular coordinates.

o **intersects**

```
public boolean intersects(Rectangle2D r)
```

Test if the Shape intersects the interior of a given Rectangle.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

o **getBounds**

```
public Rectangle getBounds()
```

Return the bounding box of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
                                    double flatness)
```

Return an iteration object that defines the boundary of the flattened shape.

Class java.awt.geom.CubicCurve2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.CubicCurve2D
           |
           +----java.awt.geom.CubicCurve2D.Float
```

---

public static class **CubicCurve2D.Float**
extends CubicCurve2D

A cubic parametric curve segment specified with float coordinates.

Variable Index

o **ctrlx1**
  The X coordinate of the first control point of the cubic curve segment.
o **ctrlx2**
  The X coordinate of the second control point of the cubic curve segment.
o **ctrly1**
  The Y coordinate of the first control point of the cubic curve segment.
o **ctrly2**
  The Y coordinate of the second control point of the cubic curve segment.
o **x1**
  The X coordinate of the start point of the cubic curve segment.
o **x2**
  The X coordinate of the end point of the cubic curve segment.
o **y1**
  The Y coordinate of the start point of the cubic curve segment.
o **y2**
  The Y coordinate of the end point of the cubic curve segment.

Constructor Index

o **CubicCurve2D.Float**()
  Constructs and initializes a CubicCurve with coordinates (0, 0, 0, 0, 0, 0).
o **CubicCurve2D.Float**(float, float, float, float, float, float, float, float)

Constructs and initializes a CubicCurve from the specified coordinates.

Method Index

o **getBounds2D**()
    Return the bounding box of the shape.
o **getCtrlX1**()
    Returns the X coordinate of the first control point in double precision.
o **getCtrlX2**()
    Returns the X coordinate of the second control point in double precision.
o **getCtrlY1**()
    Returns the Y coordinate of the first control point in double precision.
o **getCtrlY2**()
    Returns the Y coordinate of the second control point in double precision.
o **getX1**()
    Returns the X coordinate of the start point in double precision.
o **getX2**()
    Returns the X coordinate of the end point in double precision.
o **getY1**()
    Returns the Y coordinate of the start point in double precision.
o **getY2**()
    Returns the Y coordinate of the end point in double precision.
o **setCurve**(double, double, double, double, double, double, double, double)
    Sets the location of the endpoints and controlpoints of this curve to the specified double coordinates.
o **setCurve**(float, float, float, float, float, float, float, float)
    Sets the location of the endpoints and controlpoints of this curve to the specified float coordinates.

Variables

o **x1**

```
public float x1
```

The X coordinate of the start point of the cubic curve segment.

o **y1**

```
public float y1
```

The Y coordinate of the start point of the cubic curve segment.

o **ctrlx1**

```
public float ctrlx1
```

The X coordinate of the first control point of the cubic curve segment.

o **ctrly1**

```
public float ctrly1
```

The Y coordinate of the first control point of the cubic curve segment.

o **ctrlx2**

```
public float ctrlx2
```

The X coordinate of the second control point of the cubic curve segment.

o **ctrly2**

```
public float ctrly2
```

The Y coordinate of the second control point of the cubic curve segment.

o **x2**

```
public float x2
```

The X coordinate of the end point of the cubic curve segment.

o **y2**

```
public float y2
```

The Y coordinate of the end point of the cubic curve segment.

Constructors

o **CubicCurve2D.Float**

```
public CubicCurve2D.Float()
```

Constructs and initializes a CubicCurve with coordinates (0, 0, 0, 0, 0, 0).

o **CubicCurve2D.Float**

```
public CubicCurve2D.Float(float x1,
                          float y1,
                          float ctrlx1,
                          float ctrly1,
                          float ctrlx2,
                          float ctrly2,
                          float x2,
```

```
                              float y2)
```

Constructs and initializes a CubicCurve from the specified coordinates.

Methods

o **getX1**

```
public double getX1()
```

Returns the X coordinate of the start point in double precision.

Overrides:
getX1 in class CubicCurve2D

o **getY1**

```
public double getY1()
```

Returns the Y coordinate of the start point in double precision.

Overrides:
getY1 in class CubicCurve2D

o **getCtrlX1**

```
public double getCtrlX1()
```

Returns the X coordinate of the first control point in double precision.

Overrides:
getCtrlX1 in class CubicCurve2D

o **getCtrlY1**

```
public double getCtrlY1()
```

Returns the Y coordinate of the first control point in double precision.

Overrides:
getCtrlY1 in class CubicCurve2D

o **getCtrlX2**

```
public double getCtrlX2()
```

Returns the X coordinate of the second control point in double precision.

Overrides:
getCtrlX2 in class CubicCurve2D

o **getCtrlY2**

```
public double getCtrlY2()
```

Returns the Y coordinate of the second control point in double precision.

Overrides:
getCtrlY2 in class CubicCurve2D

o **getX2**

```
public double getX2()
```

Returns the X coordinate of the end point in double precision.

Overrides:
getX2 in class CubicCurve2D

o **getY2**

```
public double getY2()
```

Returns the Y coordinate of the end point in double precision.

Overrides:
getY2 in class CubicCurve2D

o **setCurve**

```
public void setCurve(double x1,
                     double y1,
                     double ctrlx1,
                     double ctrly1,
                     double ctrlx2,
                     double ctrly2,
                     double x2,
                     double y2)
```

Sets the location of the endpoints and controlpoints of this curve to the specified double coordinates.

Overrides:
setCurve in class CubicCurve2D

o **setCurve**

```
public void setCurve(float x1,
```

```
                      float y1,
                      float ctrlx1,
                      float ctrly1,
                      float ctrlx2,
                      float ctrly2,
                      float x2,
                      float y2)
```

Sets the location of the endpoints and controlpoints of this curve to the specified float coordinates.

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the bounding box of the shape.

Class java.awt.geom.Dimension2D

```
java.lang.Object
   |
   +----java.awt.geom.Dimension2D
```

---

public abstract class **Dimension2D**
extends Object

A class to encapsulate a width and a height Dimension.

This class is only the abstract superclass for all objects which store a 2D dimension. The actual storage representation of the sizes is left to the subclass.

Constructor Index

o **Dimension2D**()

Method Index

o **getHeight**()
  Returns the height of this dimension in double precision.
o **getWidth**()
  Returns the width of this dimension in double precision.
o **setSize**(Dimension2D)
  Set the size of this Dimension object to match the specified size.
o **setSize**(double, double)
  Set the size of this Dimension object to the specified width and height.

Constructors

o **Dimension2D**

```
protected Dimension2D()
```

Methods

o **getWidth**

```
public abstract double getWidth()
```

Returns the width of this dimension in double precision.

o **getHeight**

```
public abstract double getHeight()
```

Returns the height of this dimension in double precision.

o **setSize**

```
public abstract void setSize(double width,
                             double height)
```

Set the size of this Dimension object to the specified width and height. This method is included for completeness, to parallel the getSize method of Component.

Parameters:
width – the new width for the Dimension object
height – the new height for the Dimension object

o **setSize**

```
public void setSize(Dimension2D d)
```

Set the size of this Dimension object to match the specified size. This method is included for completeness, to parallel the getSize method of Component.

Parameters:
d – the new size for the Dimension object

Class java.awt.geom.Ellipse2D

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Ellipse2D
```

public abstract class **Ellipse2D**
extends RectangularShape
super class of:
    Ellipse2D.Float

An ellipse defined by a bounding rectangle.

This class is only the abstract superclass for all objects which store a 2D ellipse. The actual storage representation of the coordinates is left to the subclass.

Class Index

o **Ellipse2D.Float**
    An ellipse specified in float precision.

Constructor Index

o **Ellipse2D**()

Method Index

o **contains**(double, double)
    Test if a given Point is inside the boundary of the shape.
o **contains**(double, double, double, double)
    Test if the interior of the Shape entirely contains the given Rectangle.
o **getPathIterator**(AffineTransform)
    Return an iteration object that defines the boundary of the shape.
o **intersects**(double, double, double, double)

Test if the interior of the Shape intersects the interior of a given Rectangle.

Constructors

o **Ellipse2D**

```
protected Ellipse2D()
```

Methods

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the interior of the Shape intersects the interior of a given Rectangle.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given Rectangle.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

Class java.awt.geom.Ellipse2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Ellipse2D
                   |
                   +----java.awt.geom.Ellipse2D.Float
```

public static class **Ellipse2D.Float**
extends Ellipse2D

An ellipse specified in float precision.

Variable Index

- **height**
  The overall height of the ellipse.
- **width**
  The overall width of the ellipse.
- **x**
  The x coordinate of the upper left corner of the ellipse.
- **y**
  The y coordinate of the upper left corner of the ellipse.

Constructor Index

- **Ellipse2D.Float**()
  Constructs a new ellipse, initialized to location (0, 0) and size (0, 0).
- **Ellipse2D.Float**(float, float, float, float)
  Constructs and initializes an ellipse from the specified coordinates.

Method Index

o **getBounds2D**()
    Return the high precision bounding box of the shape.
o **getHeight**()
    Returns the overall height of the ellipse in double precision.
o **getWidth**()
    Returns the overall width of the ellipse in double precision.
o **getX**()
    Returns the X coordinate of the upper left corner of the ellipse in double precision.
o **getY**()
    Returns the Y coordinate of the upper left corner of the ellipse in double precision.
o **isEmpty**()
    Determines whether the rectangular shape is empty.
o **setBounds**(double, double, double, double)
    Sets the location and size of this ellipse to the specified float values.
o **setBounds**(float, float, float, float)
    Sets the location and size of this ellipse to the specified float values.

Variables

o **x**

```
public float x
```

The x coordinate of the upper left corner of the ellipse.

o **y**

```
public float y
```

The y coordinate of the upper left corner of the ellipse.

o **width**

```
public float width
```

The overall width of the ellipse.

o **height**

```
public float height
```

The overall height of the ellipse.

Constructors

o **Ellipse2D.Float**

```
public Ellipse2D.Float()
```

Constructs a new ellipse, initialized to location (0, 0) and size (0, 0).

o **Ellipse2D.Float**

```
public Ellipse2D.Float(float x,
                       float y,
                       float w,
                       float h)
```

Constructs and initializes an ellipse from the specified coordinates.

Parameters:
x – the x coordinate
y – the y coordinate
width – the width of the rectangle
height – the height of the rectangle

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the upper left corner of the ellipse in double precision.

Overrides:
getX in class RectangularShape

o **getY**

```
public double getY()
```

Returns the Y coordinate of the upper left corner of the ellipse in double precision.

Overrides:
getY in class RectangularShape

o **getWidth**

```
public double getWidth()
```

Returns the overall width of the ellipse in double precision.

Overrides:
getWidth in class RectangularShape

o **getHeight**

```
public double getHeight()
```

Returns the overall height of the ellipse in double precision.

Overrides:
getHeight in class RectangularShape

o **isEmpty**

```
public boolean isEmpty()
```

Determines whether the rectangular shape is empty.

Overrides:
isEmpty in class RectangularShape

o **setBounds**

```
public void setBounds(float x,
                      float y,
                      float w,
                      float h)
```

Sets the location and size of this ellipse to the specified float values.

o **setBounds**

```
public void setBounds(double x,
                      double y,
                      double w,
                      double h)
```

Sets the location and size of this ellipse to the specified float values.

Overrides:
setBounds in class RectangularShape

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

Class java.awt.geom.FlatteningPathIterator

```
java.lang.Object
   |
   +----java.awt.geom.FlatteningPathIterator
```

---

public class **FlatteningPathIterator**
extends Object
implements PathIterator

This class returns a flattened view of another PathIterator object. Other Shape classes can use this class to provide flattening behavior for their paths without having to perform the interpolation calculations themselves.

Constructor Index

o **FlatteningPathIterator**(PathIterator, double)
    Constructs a new FlatteningPathIterator object that flattens a path as it iterates over it.

Method Index

o **currentSegment**(double[])
    Returns the coordinates and type of the current path segment in the iteration.
o **currentSegment**(float[])
    Returns the coordinates and type of the current path segment in the iteration.
o **getWindingRule**()
    Return the winding rule for determining the interior of the path.
o **isDone**()
    Tests if there are more points to read.
o **next**()
    Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction.

Constructors

o **FlatteningPathIterator**

```
public FlatteningPathIterator(PathIterator src,
                              double flatness)
```

Constructs a new FlatteningPathIterator object that flattens a path as it iterates over it.

Parameters:
src – the original unflattened path being iterated over
flatness – the maximum allowable distance between the control points

Methods

o **getWindingRule**

```
public int getWindingRule()
```

Return the winding rule for determining the interior of the path.

See Also:
WIND_EVEN_ODD, WIND_NON_ZERO

o **isDone**

```
public boolean isDone()
```

Tests if there are more points to read.

Returns:
true if there are more points to read

o **next**

```
public void next()
```

Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction.

o **currentSegment**

```
public int currentSegment(float coords[])
```

Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, or SEG_CLOSE. A float array of length 6 must be passed in and may be used to store the coordinates of the point(s). Each point is stored as a pair of float x,y coordinates. SEG_MOVETO and SEG_LINETO types will return one point, and SEG_CLOSE will not return any points.

See Also:
SEG_MOVETO, SEG_LINETO, SEG_CLOSE

o **currentSegment**

```
public int currentSegment(double coords[])
```

Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, or SEG_CLOSE. A double array of length 6 must be passed in and may be used to store the coordinates of the point(s). Each point is stored as a pair of double x,y coordinates. SEG_MOVETO and SEG_LINETO types will return one point, and SEG_CLOSE will not return any points.

See Also:
SEG_MOVETO, SEG_LINETO, SEG_CLOSE

Class java.awt.geom.GeneralPath

```
java.lang.Object
   |
   +----java.awt.geom.GeneralPath
```

---

public final class **GeneralPath**
extends Object
implements Shape

This class represents a geometric path constructed from straight lines, quadratic and cubic (Bezier) curves. It can contain multiple subpaths.

The winding rule specifies how the interior of a path will be determined. There are two types of winding rules: EVEN_ODD and NON_ZERO. An EVEN−ODD winding rule means that enclosed regions of the path alternate between interior and exterior areas as traversed from the outside of the path towards a point inside the region. For a NON_ZERO winding rule, we start by conceptually drawing a ray from a given point to infinity in any direction and then by examining all of the places where the path intersects the ray. We now traverse the path in one direction and keep track of the number of times that the path crosses the ray from left to right, and the number of times that the path crosses the ray from right to left. If these numbers are equal, the point is outside of the path. If they are not equal, the point is inside.

Variable Index

o **EVEN_ODD**
  An even−odd winding rule for determining the interior of a path.
o **NON_ZERO**
  A non−zero winding rule for determining the interior of a path.

Constructor Index

o **GeneralPath**(int)
  Constructs a new GeneralPath object.
o **GeneralPath**(int, int)
  Constructs a new GeneralPath object with the specified initial capacity to store path

coordinates.

Method Index

o **append**(PathIterator, boolean)
  Appends the geometry of the specified PathIterator object to the path, possibly connecting the geometry to the existing path segments with a line segment.
o **append**(Shape, boolean)
  Appends the geometry of the specified Shape object to the path, possibly connecting the geometry to the existing path segments with a line segment.
o **closePath**()
  Closes the current subpath by drawing a straight line back to the coordinates of the last moveTo.
o **contains**(double, double)
  Test if a given coordinate is inside the boundary of the shape.
o **contains**(double, double, double, double)
  Test if a given coordinate is inside the boundary of the shape.
o **contains**(Point2D)
  Test if a given Point is inside the boundary of the shape.
o **contains**(Rectangle2D)
  Test if a given Point is inside the boundary of the shape.
o **createTransformedShape**(AffineTransform)
  Returns a new transformed Path.
o **curveTo**(float, float, float, float, float, float)
  Adds 3 points to the path by drawing a Bezier curve from the current coordinates through the third set of specified coordinates, using the first and second sets of specified coordinates as Bezier control points.
o **getBounds**()
  Return the bounding box of the shape.
o **getBounds2D**()
  Returns the bounding box of the path.
o **getCurrentPoint**()
  Returns the most recent point added to the end of the path.
o **getPathIterator**(AffineTransform)
  Return an iterator object that iterates along the boundary of the shape and provides access to the geometry of the outline of the shape.
o **getPathIterator**(AffineTransform, double)
  Return an iterator object that iterates along the boundary of the flattened shape and provides access to the geometry of the outline of the shape.
o **getWindingRule**()
  Returns the fill style winding rule.
o **intersects**(double, double, double, double)
  Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.
o **intersects**(Rectangle2D)
  Test if the interior of the Shape intersects the interior of a given Rectangle.
o **lineTo**(float, float)
  Adds a point to the path by drawing a straight line from the current coordinates to the new specified coordinates.

o **moveTo**(float, float)
  Adds a point to the path by moving to the specified coordinates.
o **quadTo**(float, float, float, float)
  Adds 2 points to the path by drawing a Quadratic curve from the current coordinates through the second set of specified coordinates, using the first set of specified coordinates as a quadratic parametric control point.
o **reset**()
  Resets the path to empty.
o **setWindingRule**(int)
  Sets the winding rule for this path to the specified value.

Variables

o **EVEN_ODD**

```
public static final byte EVEN_ODD
```

An even−odd winding rule for determining the interior of a path.

o **NON_ZERO**

```
public static final byte NON_ZERO
```

A non−zero winding rule for determining the interior of a path.

Constructors

o **GeneralPath**

```
public GeneralPath(int rule)
```

Constructs a new GeneralPath object.

Parameters:
  rule − The winding rule.

o **GeneralPath**

```
public GeneralPath(int rule,
                   int initialCapacity)
```

Constructs a new GeneralPath object with the specified initial capacity to store path coordinates. This number is an initial guess as to how many coordinates will be in the path, but the storage will be expanded as needed to store whatever path segments are added to this path.

Parameters:
  rule − The winding rule.

initialCapacity – The estimate for the number of coordinates in the path.

Methods

o **moveTo**

```
public synchronized void moveTo(float x,
                                float y)
```

Adds a point to the path by moving to the specified coordinates.

o **lineTo**

```
public synchronized void lineTo(float x,
                                float y)
```

Adds a point to the path by drawing a straight line from the current coordinates to the new specified coordinates.

o **quadTo**

```
public synchronized void quadTo(float x1,
                                float y1,
                                float x2,
                                float y2)
```

Adds 2 points to the path by drawing a Quadratic curve from the current coordinates through the second set of specified coordinates, using the first set of specified coordinates as a quadratic parametric control point.

Parameters:

x1 – the X coordinate of the first quadratic control point
y1 – the Y coordinate of the first quadratic control point
x2 – the X coordinate of the final endpoint
y2 – the Y coordinate of the final endpoint

o **curveTo**

```
public synchronized void curveTo(float x1,
                                 float y1,
                                 float x2,
                                 float y2,
                                 float x3,
                                 float y3)
```

Adds 3 points to the path by drawing a Bezier curve from the current coordinates through the third set of specified coordinates, using the first and second sets of specified coordinates as Bezier control points.

Parameters:

x1 – the X coordinate of the first Bezier control point
y1 – the Y coordinate of the first Bezier control point
x2 – the X coordinate of the second Bezier control point
y2 – the Y coordinate of the second Bezier control point
x3 – the X coordinate of the final endpoint
y3 – the Y coordinate of the final endpoint

o **closePath**

```
public synchronized void closePath()
```

Closes the current subpath by drawing a straight line back to the coordinates of the last moveTo. If the path is already closed, then this method has no effect.

o **append**

```
public void append(Shape s,
                   boolean connect)
```

Appends the geometry of the specified Shape object to the path, possibly connecting the geometry to the existing path segments with a line segment. If the connect parameter is true and the path is not empty, then any initial moveTo in the geometry of the appended Shape will be turned into a lineTo segment. If the destination coordinates of such a connecting lineTo segment match the ending coordinates of a currently open subpath, then the segment will be omitted as superfluous. The winding rule of the specified Shape will be ignored and the appended geometry will be governed by the winding rule specified for this path.

Parameters:
s – the shape whose geometry will be appended to this path
connect – a boolean to control whether or not to turn an initial moveTo segment into a lineTo segment to connect the new geometry to the existing path

o **append**

```
public void append(PathIterator pi,
                   boolean connect)
```

Appends the geometry of the specified PathIterator object to the path, possibly connecting the geometry to the existing path segments with a line segment. If the connect parameter is true and the path is not empty, then any initial moveTo in the geometry of the appended Shape will be turned into a lineTo segment. If the destination coordinates of such a connecting lineTo segment match the ending coordinates of a currently open subpath, then the segment will be omitted as superfluous. The winding rule of the specified Shape will be ignored and the appended geometry will be governed by the winding rule specified for this path.

Parameters:
s – the shape whose geometry will be appended to this path
connect – a boolean to control whether or not to turn an initial moveTo segment into a lineTo segment to connect the new geometry to the existing path

o **getWindingRule**

```
public synchronized int getWindingRule()
```

Returns the fill style winding rule.

o **setWindingRule**

```
public void setWindingRule(int rule)
```

Sets the winding rule for this path to the specified value.

o **getCurrentPoint**

```
public synchronized Point2D getCurrentPoint()
```

Returns the most recent point added to the end of the path.

Returns:
a Point object containing the ending coordinate of the path or null if there are no points in the path.

o **reset**

```
public synchronized void reset()
```

Resets the path to empty. The append position is set back to the beginning of the path and all coordinates and point types are forgotten.

o **createTransformedShape**

```
public synchronized Shape createTransformedShape(AffineTransform t)
```

Returns a new transformed Path.

o **getBounds**

```
public Rectangle getBounds()
```

Return the bounding box of the shape.

o **getBounds2D**

```
public synchronized Rectangle2D getBounds2D()
```

Returns the bounding box of the path.

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **contains**

```
public boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if a given coordinate is inside the boundary of the shape.

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.

o **intersects**

```
public boolean intersects(Rectangle2D r)
```

Test if the interior of the Shape intersects the interior of a given Rectangle.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iterator object that iterates along the boundary of the shape and provides access to the geometry of the outline of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
                                    double flatness)
```

Return an iterator object that iterates along the boundary of the flattened shape and provides access to the geometry of the outline of the shape.

All Packages Class Hierarchy This Package Previous Next Index

Class java.awt.geom.GeneralPathIterator

```
java.lang.Object
   |
   +----java.awt.geom.GeneralPathIterator
```

public class **GeneralPathIterator**
extends Object
implements PathIterator

This class represents the iterator for General Paths. It can be used to retrieve all of the elements in a GeneralPath. The GeneralPath.getPathIterator() method is used to create a GeneralPathIterator for a particular GeneralPath. The iterator can be used to iterator the path only once. Subsequent iterations require a new iterator.

See Also:
GeneralPath

Method Index

- **currentSegment**(double[])
  Returns the coordinates and type of the current path segment in the iteration.
- **currentSegment**(float[])
  Returns the coordinates and type of the current path segment in the iteration.
- **getWindingRule**()
  Return the winding rule for determining the interior of the path.
- **isDone**()
  Tests if there are more points to read.
- **next**()
  Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction.

Methods

o **getWindingRule**

```
public int getWindingRule()
```

Return the winding rule for determining the interior of the path.

See Also:
WIND_EVEN_ODD, WIND_NON_ZERO

o **isDone**

```
public boolean isDone()
```

Tests if there are more points to read.

Returns:
true if there are more points to read

o **next**

```
public void next()
```

Moves the iterator to the next segment of the path forwards along the primary direction of traversal as long as there are more points in that direction.

o **currentSegment**

```
public int currentSegment(float coords[])
```

Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, or SEG_CLOSE. A float array of length 6 must be passed in and may be used to store the coordinates of the point(s). Each point is stored as a pair of float x,y coordinates. SEG_MOVETO and SEG_LINETO types will return one point, SEG_QUADTO will return two points, SEG_CUBICTO will return 3 points and SEG_CLOSE will not return any points.

See Also:
SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, SEG_CLOSE

o **currentSegment**

```
public int currentSegment(double coords[])
```

Returns the coordinates and type of the current path segment in the iteration. The return value is the path segment type: SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, or SEG_CLOSE. A double array of length 6 must be passed in and may be used to store the coordinates of the point(s). Each point is stored as a pair of double x,y coordinates. SEG_MOVETO and SEG_LINETO types will return one point, SEG_QUADTO will return two points, SEG_CUBICTO will return 3 points and SEG_CLOSE will not return any points.

See Also:

SEG_MOVETO, SEG_LINETO, SEG_QUADTO, SEG_CUBICTO, SEG_CLOSE

Class java.awt.geom.Line2D

```
java.lang.Object
   |
   +----java.awt.geom.Line2D
```

---

public abstract class **Line2D**
extends Object
implements Shape
super class of:
Line2D.Float

A line segment in (x, y) coordinate space. For the purposes of the Shape interface, a coordinate is considered to be inside or contained by this line segment if the coordinate lies exactly on the line segment.

This class is only the abstract superclass for all objects which store a 2D line segment. The actual storage representation of the coordinates is left to the subclass.

Class Index

o **Line2D.Float**
A line segment specified with float coordinates.

Constructor Index

o **Line2D**()

Method Index

o **colinear**(double, double)
Test if a given (x, y) coordinate is colinear with the line segment.
o **colinear**(Point2D)
Test if a given Point is colinear with the line segment.
o **contains**(double, double)
Test if a given coordinate is inside the boundary of the shape.

o **contains**(double, double, double, double)
    Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
o **contains**(Point2D)
    Test if a given Point is inside the boundary of the shape.
o **contains**(Rectangle2D)
    Test if the interior of the Shape entirely contains the given Rectangle.
o **getBounds**()
    Return the bounding box of the shape.
o **getPathIterator**(AffineTransform)
    Return an iteration object that defines the boundary of the shape.
o **getPathIterator**(AffineTransform, double)
    Return an iteration object that defines the boundary of the flattened shape.
o **getX1**()
    Returns the X coordinate of the start point in double precision.
o **getX2**()
    Returns the X coordinate of the end point in double precision.
o **getY1**()
    Returns the Y coordinate of the start point in double precision.
o **getY2**()
    Returns the Y coordinate of the end point in double precision.
o **intersects**(double, double, double, double)
    Test if the Shape intersects the interior of a given set of rectangular coordinates.
o **intersects**(Rectangle2D)
    Test if the Shape intersects the interior of a given Rectangle.
o **intersectsLine**(double, double, double, double)
    Tests if the line segment from (x1, y1) to (x2, y2) intersects this line segment.
o **intersectsLine**(Line2D)
    Tests if the given line segment intersects this line segment.
o **linesIntersect**(double, double, double, double, double, double, double, double)
    Tests if the line segment from (x1, y1) to (x2, y2) intersects the line segment from (x3, y3) to (x4, y4).
o **ptSegDist**(double, double, double, double, double, double)
    Returns the distance from a point to a line segment.
o **ptSegDistSq**(double, double, double, double, double, double)
    Returns the square of the distance from a point to a line segment.
o **relativeCCW**(double, double)
    Return an indicator of where the specified point (x, y) lies with respect to this line segment.
o **relativeCCW**(double, double, double, double, double, double)
    Return an indicator of where the specified point (px, py) lies with respect to the line segment from (x1, y1) to (x2, y2).
o **relativeCCW**(Point2D)
    Return an indicator of where the specified point lies with respect to this line segment.
o **setLine**(double, double, double, double)
    Sets the location of the endpoints of this line to the specified double coordinates.
o **setLine**(Line2D)
    Sets the location of the endpoints of this line to the same as those in the specified Line.
o **setLine**(Point2D, Point2D)
    Sets the location of the endpoints of this line to the specified Point coordinates.

Constructors

o **Line2D**

```
protected Line2D()
```

Methods

o **getX1**

```
public abstract double getX1()
```

Returns the X coordinate of the start point in double precision.

o **getY1**

```
public abstract double getY1()
```

Returns the Y coordinate of the start point in double precision.

o **getX2**

```
public abstract double getX2()
```

Returns the X coordinate of the end point in double precision.

o **getY2**

```
public abstract double getY2()
```

Returns the Y coordinate of the end point in double precision.

o **setLine**

```
public abstract void setLine(double x1,
                             double y1,
                             double x2,
                             double y2)
```

Sets the location of the endpoints of this line to the specified double coordinates.

o **setLine**

```
public void setLine(Point2D p1,
                    Point2D p2)
```

Sets the location of the endpoints of this line to the specified Point coordinates.

o **setLine**

```
public void setLine(Line2D l)
```

Sets the location of the endpoints of this line to the same as those in the specified Line.

o **colinear**

```
public boolean colinear(double x,
                        double y)
```

Test if a given (x, y) coordinate is colinear with the line segment.

o **colinear**

```
public boolean colinear(Point2D p)
```

Test if a given Point is colinear with the line segment.

o **relativeCCW**

```
public static int relativeCCW(double x1,
                              double y1,
                              double x2,
                              double y2,
                              double px,
                              double py)
```

Return an indicator of where the specified point (px, py) lies with respect to the line segment from (x1, y1) to (x2, y2). The value will be 1 if the line segment must turn counterclockwise to point at the specified point, −1 if it must turn clockwise, or 0 if the point lies exactly on the line segment. If the point is colinear with the line segment, but not between the endpoints, then the value will be −1 if the point lies "beyond (x1, y1)" or 1 if the point lies "beyond (x2, y2)".

o **relativeCCW**

```
public int relativeCCW(double x,
                       double y)
```

Return an indicator of where the specified point (x, y) lies with respect to this line segment. The value will be 1 if the line segment must turn counterclockwise to point at the specified point, −1 if it must turn clockwise, or 0 if the point lies exactly on the line segment. If the point is colinear with the line segment, but not between the endpoints, then the value will be −1 if the point lies "beyond (x1, y1)" or 1 if the point lies "beyond (x2, y2)".

o **relativeCCW**

```
public int relativeCCW(Point2D p)
```

Return an indicator of where the specified point lies with respect to this line segment. The value will be 1 if the line segment must turn counterclockwise to point at the specified point, −1 if it must turn clockwise, or 0 if the point lies exactly on the line segment. If the point is colinear with the line segment, but not between the endpoints, then the value will be −1 if the point lies "beyond (x1, y1)" or 1 if the point lies "beyond (x2, y2)".

o **linesIntersect**

```
public static boolean linesIntersect(double x1,
                                     double y1,
                                     double x2,
                                     double y2,
                                     double x3,
                                     double y3,
                                     double x4,
                                     double y4)
```

Tests if the line segment from (x1, y1) to (x2, y2) intersects the line segment from (x3, y3) to (x4, y4).

o **intersectsLine**

```
public boolean intersectsLine(double x1,
                              double y1,
                              double x2,
                              double y2)
```

Tests if the line segment from (x1, y1) to (x2, y2) intersects this line segment.

o **intersectsLine**

```
public boolean intersectsLine(Line2D l)
```

Tests if the given line segment intersects this line segment.

o **ptSegDistSq**

```
public static double ptSegDistSq(double x0,
                                 double y0,
                                 double x1,
                                 double y1,
                                 double px,
                                 double py)
```

Returns the square of the distance from a point to a line segment.

Parameters:

x0 – the x coordinate of the beginning of the line segment
y0 – the y coordinate of the beginning of the line segment
x1 – the x coordinate of the end of the line segment
y1 – the y coordinate of the end of the line segment
px – the x coordinate of the point being measured
py – the y coordinate of the point being measured

o **ptSegDist**

```
public static double ptSegDist(double x0,
                               double y0,
                               double x1,
                               double y1,
                               double px,
                               double py)
```

Returns the distance from a point to a line segment.

Parameters:
x0 – the x coordinate of the beginning of the line segment
y0 – the y coordinate of the beginning of the line segment
x1 – the x coordinate of the end of the line segment
y1 – the y coordinate of the end of the line segment
px – the x coordinate of the point being measured
py – the y coordinate of the point being measured

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **contains**

```
public boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the Shape intersects the interior of a given set of rectangular coordinates.

o **intersects**

```
public boolean intersects(Rectangle2D r)
```

Test if the Shape intersects the interior of a given Rectangle.

o **contains**

```
public boolean contains(double x,
                        double y,
```

```
                         double w,
                         double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

o **getBounds**

```
public Rectangle getBounds()
```

Return the bounding box of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
                                    double flatness)
```

Return an iteration object that defines the boundary of the flattened shape.

Class java.awt.geom.Line2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.Line2D
           |
           +----java.awt.geom.Line2D.Float
```

---

public static class **Line2D.Float**
extends Line2D

A line segment specified with float coordinates.

Variable Index

o **x1**
The X coordinate of the start point of the line segment.
o **x2**
The X coordinate of the end point of the line segment.
o **y1**
The Y coordinate of the start point of the line segment.
o **y2**
The Y coordinate of the end point of the line segment.

Constructor Index

o **Line2D.Float**()
Constructs and initializes a Line with coordinates (0, 0) -> (0, 0).
o **Line2D.Float**(float, float, float, float)
Constructs and initializes a Line from the specified coordinates.
o **Line2D.Float**(Point2D, Point2D)
Constructs and initializes a Line from the specified Points.

Method Index

o **getBounds2D**()
Return the high precision bounding box of the shape.
o **getX1**()
Returns the X coordinate of the start point in double precision.
o **getX2**()
Returns the X coordinate of the end point in double precision.
o **getY1**()
Returns the Y coordinate of the start point in double precision.
o **getY2**()
Returns the Y coordinate of the end point in double precision.
o **setLine**(double, double, double, double)
Sets the location of the endpoints of this line to the specified double coordinates.

Variables

o **x1**

```
public float x1
```

The X coordinate of the start point of the line segment.

o **y1**

```
public float y1
```

The Y coordinate of the start point of the line segment.

o **x2**

```
public float x2
```

The X coordinate of the end point of the line segment.

o **y2**

```
public float y2
```

The Y coordinate of the end point of the line segment.

Constructors

o **Line2D.Float**

```
public Line2D.Float()
```

Constructs and initializes a Line with coordinates (0, 0) -> (0, 0).

o **Line2D.Float**

```
public Line2D.Float(float x1,
                    float y1,
                    float x2,
                    float y2)
```

Constructs and initializes a Line from the specified coordinates.

o **Line2D.Float**

```
public Line2D.Float(Point2D p1,
                    Point2D p2)
```

Constructs and initializes a Line from the specified Points.

Methods

o **getX1**

```
public double getX1()
```

Returns the X coordinate of the start point in double precision.

Overrides:
getX1 in class Line2D

o **getY1**

```
public double getY1()
```

Returns the Y coordinate of the start point in double precision.

Overrides:
getY1 in class Line2D

o **getX2**

```
public double getX2()
```

Returns the X coordinate of the end point in double precision.

Overrides:
getX2 in class Line2D

o **getY2**

```
public double getY2()
```

Returns the Y coordinate of the end point in double precision.

Overrides:
getY2 in class Line2D

o **setLine**

```
public void setLine(double x1,
                    double y1,
                    double x2,
                    double y2)
```

Sets the location of the endpoints of this line to the specified double coordinates.

Overrides:
setLine in class Line2D

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

Class java.awt.geom.Point2D

```
java.lang.Object
   |
   +----java.awt.geom.Point2D
```

public abstract class **Point2D**
extends Object
super class of:
Point2D.Double, Point2D.Float

A point representing a location in (x, y) coordinate space.

This class is only the abstract superclass for all objects which store a 2D coordinate. The actual storage representation of the coordinates is left to the subclass.

Class Index

o **Point2D.Double**
A point specified in double precision.
o **Point2D.Float**
A point specified in float precision.

Constructor Index

o **Point2D**()

Method Index

o **equals**(Object)
Determines whether two points are equal.
o **getX**()
Returns the X coordinate of the point in double precision.
o **getY**()
Returns the Y coordinate of the point in double precision.
o **setLocation**(double, double)

Sets the location of this point to the specified float coordinates.

o **setLocation**(Point2D)

Sets the location of this point to the same coordinates as the specified Point object.

Constructors

o **Point2D**

```
protected Point2D()
```

Methods

o **getX**

```
public abstract double getX()
```

Returns the X coordinate of the point in double precision.

o **getY**

```
public abstract double getY()
```

Returns the Y coordinate of the point in double precision.

o **setLocation**

```
public abstract void setLocation(double x,
                                 double y)
```

Sets the location of this point to the specified float coordinates.

o **setLocation**

```
public void setLocation(Point2D p)
```

Sets the location of this point to the same coordinates as the specified Point object.

o **equals**

```
public boolean equals(Object obj)
```

Determines whether two points are equal. Two instances of `Point2D` are equal if the values of their `x` and `y` member fields, representing their position in the coordinate space, are the same.

Parameters:
obj – an object to be compared with this point.

Returns:
`true` if the object to be compared is an instance of `Point2D` and has the same values;

`false` otherwise.

Overrides:

equals in class Object

Class java.awt.geom.Point2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.Point2D
           |
           +----java.awt.geom.Point2D.Float
```

public static class **Point2D.Float**
extends Point2D

A point specified in float precision.

Variable Index

o **x**
The X coordinate.
o **y**
The Y coordinate.

Constructor Index

o **Point2D.Float**()
Constructs and initializes a Point with coordinates (0, 0).
o **Point2D.Float**(float, float)
Constructs and initializes a Point with the specified coordinates.

Method Index

o **getX**()
Returns the X coordinate of the point in double precision.
o **getY**()
Returns the Y coordinate of the point in double precision.
o **setLocation**(double, double)
Sets the location of this point to the specified double coordinates.
o **setLocation**(float, float)

Sets the location of this point to the specified float coordinates.

o **toString**()

Returns a String that represents the value of this Object.

Variables

o **x**

```
public float x
```

The X coordinate.

o **y**

```
public float y
```

The Y coordinate.

Constructors

o **Point2D.Float**

```
public Point2D.Float()
```

Constructs and initializes a Point with coordinates (0, 0).

o **Point2D.Float**

```
public Point2D.Float(float x,
                     float y)
```

Constructs and initializes a Point with the specified coordinates.

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the point in double precision.

Overrides:

getX in class Point2D

o **getY**

```
public double getY()
```

Returns the Y coordinate of the point in double precision.

Overrides:
getY in class Point2D

o **setLocation**

```
public void setLocation(double x,
                        double y)
```

Sets the location of this point to the specified double coordinates.

Overrides:
setLocation in class Point2D

o **setLocation**

```
public void setLocation(float x,
                        float y)
```

Sets the location of this point to the specified float coordinates.

o **toString**

```
public String toString()
```

Returns a String that represents the value of this Object.

Overrides:
toString in class Object

Class java.awt.geom.Point2D.Double

```
java.lang.Object
   |
   +----java.awt.geom.Point2D
           |
           +----java.awt.geom.Point2D.Double
```

public static class **Point2D.Double**
extends Point2D

A point specified in double precision.

Variable Index

o **x**
    The x coordinate of the point.
o **y**
    The y coordinate of the point.

Constructor Index

o **Point2D.Double**()
    Constructs and initializes a Point with coordinates (0, 0).
o **Point2D.Double**(double, double)
    Constructs and initializes a Point with the specified coordinates.

Method Index

o **getX**()
    Returns the X coordinate of the point in double precision.
o **getY**()
    Returns the Y coordinate of the point in double precision.
o **setLocation**(double, double)
    Sets the location of this point to the specified double coordinates.
o **toString**()

Returns a String that represents the value of this Object.

Variables

o **x**

```
public double x
```

The x coordinate of the point.

o **y**

```
public double y
```

The y coordinate of the point.

Constructors

o **Point2D.Double**

```
public Point2D.Double()
```

Constructs and initializes a Point with coordinates (0, 0).

o **Point2D.Double**

```
public Point2D.Double(double x,
                      double y)
```

Constructs and initializes a Point with the specified coordinates.

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the point in double precision.

Overrides:
getX in class Point2D

o **getY**

```
public double getY()
```

Returns the Y coordinate of the point in double precision.

Overrides:
getY in class Point2D

o **setLocation**

```
public void setLocation(double x,
                        double y)
```

Sets the location of this point to the specified double coordinates.

Overrides:
setLocation in class Point2D

o **toString**

```
public String toString()
```

Returns a String that represents the value of this Object.

Overrides:
toString in class Object

Class java.awt.geom.QuadCurve2D

```
java.lang.Object
   |
   +----java.awt.geom.QuadCurve2D
```

---

public abstract class **QuadCurve2D**
extends Object
implements Shape
super class of:
    QuadCurve2D.Float

A quadratic parametric curve segment in (x, y) coordinate space.

This class is only the abstract superclass for all objects which store a 2D quadratic curve segment. The actual storage representation of the coordinates is left to the subclass.

Class Index

o **QuadCurve2D.Float**
    A quadratic parametric curve segment specified with float coordinates.

Constructor Index

o **QuadCurve2D**()

Method Index

o **contains**(double, double)
    Test if a given coordinate is inside the boundary of the shape.
o **contains**(double, double, double, double)
    Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
o **contains**(Point2D)
    Test if a given Point is inside the boundary of the shape.
o **contains**(Rectangle2D)
    Test if the interior of the Shape entirely contains the given Rectangle.

o **getBounds**()
    Return the bounding box of the shape.
o **getCtrlX**()
    Returns the X coordinate of the control point in double precision.
o **getCtrlY**()
    Returns the Y coordinate of the control point in double precision.
o **getFlatness**()
    Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.
o **getFlatness**(double, double, double, double, double, double)
    Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the indicated controlpoints.
o **getFlatness**(double[], int)
    Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the controlpoints stored in the indicated array at the indicated index.
o **getFlatnessSq**()
    Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.
o **getFlatnessSq**(double, double, double, double, double, double)
    Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the indicated controlpoints.
o **getFlatnessSq**(double[], int)
    Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the controlpoints stored in the indicated array at the indicated index.
o **getPathIterator**(AffineTransform)
    Return an iteration object that defines the boundary of the shape.
o **getPathIterator**(AffineTransform, double)
    Return an iteration object that defines the boundary of the flattened shape.
o **getX1**()
    Returns the X coordinate of the start point in double precision.
o **getX2**()
    Returns the X coordinate of the end point in double precision.
o **getY1**()
    Returns the Y coordinate of the start point in double precision.
o **getY2**()
    Returns the Y coordinate of the end point in double precision.
o **intersects**(double, double, double, double)
    Test if the Shape intersects the interior of a given set of rectangular coordinates.
o **intersects**(Rectangle2D)
    Test if the Shape intersects the interior of a given Rectangle.
o **setCurve**(double, double, double, double, double, double)
    Sets the location of the endpoints and controlpoint of this curve to the specified double coordinates.
o **setCurve**(double[], int)
    Sets the location of the endpoints and controlpoints of this curve to the double coordinates at the specified offset in the specified array.
o **setCurve**(Point2D, Point2D, Point2D)

Sets the location of the endpoints and controlpoint of this curve to the specified Point coordinates.

o **setCurve**(Point2D[], int)
: Sets the location of the endpoints and controlpoints of this curve to the coordinates of the Point objects at the specified offset in the specified array.

o **setCurve**(QuadCurve2D)
: Sets the location of the endpoints and controlpoint of this curve to the same as those in the specified QuadCurve.

o **solveQuadratic**(double[])
: Solve the quadratic whose coefficients are in the eqn array and place the non−complex roots back into the array, returning the number of roots.

o **subdivide**(double[], int, double[], int, double[], int)
: Subdivides the quadratic curve specified by the the coordinates stored in the src array at indices (srcoff) through (srcoff + 5) and stores the resulting two subdivided curves into the two result arrays at the corresponding indices.

o **subdivide**(QuadCurve2D, QuadCurve2D)
: Subdivides this quadratic curve and stores the resulting two subdivided curves into the left and right curve parameters.

o **subdivide**(QuadCurve2D, QuadCurve2D, QuadCurve2D)
: Subdivides the quadratic curve specified by the src parameter and stores the resulting two subdivided curves into the left and right curve parameters.

Constructors

o **QuadCurve2D**

```
protected QuadCurve2D()
```

Methods

o **getX1**

```
public abstract double getX1()
```

Returns the X coordinate of the start point in double precision.

o **getY1**

```
public abstract double getY1()
```

Returns the Y coordinate of the start point in double precision.

o **getCtrlX**

```
public abstract double getCtrlX()
```

Returns the X coordinate of the control point in double precision.

o **getCtrlY**

```
public abstract double getCtrlY()
```

Returns the Y coordinate of the control point in double precision.

o **getX2**

```
public abstract double getX2()
```

Returns the X coordinate of the end point in double precision.

o **getY2**

```
public abstract double getY2()
```

Returns the Y coordinate of the end point in double precision.

o **setCurve**

```
public abstract void setCurve(double x1,
                              double y1,
                              double ctrlx,
                              double ctrly,
                              double x2,
                              double y2)
```

Sets the location of the endpoints and controlpoint of this curve to the specified double coordinates.

o **setCurve**

```
public void setCurve(double coords[],
                     int offset)
```

Sets the location of the endpoints and controlpoints of this curve to the double coordinates at the specified offset in the specified array.

o **setCurve**

```
public void setCurve(Point2D p1,
                     Point2D cp,
                     Point2D p2)
```

Sets the location of the endpoints and controlpoint of this curve to the specified Point coordinates.

o **setCurve**

```
public void setCurve(Point2D pts[],
```

```
                    int offset)
```

Sets the location of the endpoints and controlpoints of this curve to the coordinates of the Point objects at the specified offset in the specified array.

o **setCurve**

```
public void setCurve(QuadCurve2D c)
```

Sets the location of the endpoints and controlpoint of this curve to the same as those in the specified QuadCurve.

o **getFlatnessSq**

```
public static double getFlatnessSq(double x1,
                                   double y1,
                                   double ctrlx,
                                   double ctrly,
                                   double x2,
                                   double y2)
```

Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the indicated controlpoints.

o **getFlatness**

```
public static double getFlatness(double x1,
                                 double y1,
                                 double ctrlx,
                                 double ctrly,
                                 double x2,
                                 double y2)
```

Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the indicated controlpoints.

o **getFlatnessSq**

```
public static double getFlatnessSq(double coords[],
                                   int offset)
```

Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the controlpoints stored in the indicated array at the indicated index.

o **getFlatness**

```
public static double getFlatness(double coords[],
                                 int offset)
```

Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of the quadratic curve specified by the controlpoints stored in the indicated array at the indicated index.

o **getFlatnessSq**

```
public double getFlatnessSq()
```

Returns the square of the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.

o **getFlatness**

```
public double getFlatness()
```

Returns the flatness, or maximum distance of a controlpoint from the line connecting the endpoints, of this curve.

o **subdivide**

```
public void subdivide(QuadCurve2D left,
                      QuadCurve2D right)
```

Subdivides this quadratic curve and stores the resulting two subdivided curves into the left and right curve parameters. Either or both of the left and right objects may be the same as this object or null.

Parameters:

left – the quadratic curve object for storing for the left or first half of the subdivided curve

right – the quadratic curve object for storing for the right or second half of the subdivided curve

o **subdivide**

```
public static void subdivide(QuadCurve2D src,
                             QuadCurve2D left,
                             QuadCurve2D right)
```

Subdivides the quadratic curve specified by the src parameter and stores the resulting two subdivided curves into the left and right curve parameters. Either or both of the left and right objects may be the same as the src object or null.

Parameters:

src – the quadratic curve to be subdivided

left – the quadratic curve object for storing for the left or first half of the subdivided curve

right – the quadratic curve object for storing for the right or second half of the subdivided curve o **subdivide**

```
public static void subdivide(double src[],
                             int srcoff,
                             double left[],
                             int leftoff,
                             double right[],
                             int rightoff)
```

Subdivides the quadratic curve specified by the the coordinates stored in the src array at indices (srcoff) through (srcoff + 5) and stores the resulting two subdivided curves into the two result arrays at the corresponding indices. Either or both of the left and right arrays may be null or a reference to the same array and offset as the src array. Note that the last point in the first subdivided curve is the same as the first point in the second subdivided curve and thus it is possible to pass the same array for left and right and to use offsets such that rightoff equals (leftoff + 4) in order to avoid allocating extra storage for this common point.

Parameters:

src − the array holding the coordinates for the source curve
srcoff − the offset into the array of the beginning of the the 6 source coordinates
left − the array for storing the coordinates for the first half of the subdivided curve
leftoff − the offset into the array of the beginning of the the 6 left coordinates
right − the array for storing the coordinates for the second half of the subdivided curve
rightoff − the offset into the array of the beginning of the the 6 right coordinates

o **solveQuadratic**

```
public static int solveQuadratic(double eqn[])
```

Solve the quadratic whose coefficients are in the eqn array and place the non−complex roots back into the array, returning the number of roots. The quadratic solved is represented by the equation: eqn = {C, B, A}; ax^2 + bx + c = 0 A return value of −1 is used to distinguish a constant equation, which may be always 0 or never 0, from an equation which has no zeroes.

Returns:

the number of roots, or −1 if the equation is a constant

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **contains**

```
public boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the Shape intersects the interior of a given set of rectangular coordinates.

o **intersects**

```
public boolean intersects(Rectangle2D r)
```

Test if the Shape intersects the interior of a given Rectangle.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

o **getBounds**

```
public Rectangle getBounds()
```

Return the bounding box of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
                                    double flatness)
```

Return an iteration object that defines the boundary of the flattened shape.

Class java.awt.geom.QuadCurve2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.QuadCurve2D
           |
           +----java.awt.geom.QuadCurve2D.Float
```

---

public static class **QuadCurve2D.Float**
extends QuadCurve2D

A quadratic parametric curve segment specified with float coordinates.

Variable Index

o **ctrlx**
  The X coordinate of the control point of the quadratic curve segment.
o **ctrly**
  The Y coordinate of the control point of the quadratic curve segment.
o **x1**
  The X coordinate of the start point of the quadratic curve segment.
o **x2**
  The X coordinate of the end point of the quadratic curve segment.
o **y1**
  The Y coordinate of the start point of the quadratic curve segment.
o **y2**
  The Y coordinate of the end point of the quadratic curve segment.

Constructor Index

o **QuadCurve2D.Float**()
  Constructs and initializes a QuadCurve with coordinates (0, 0, 0, 0, 0, 0).
o **QuadCurve2D.Float**(float, float, float, float, float, float)
  Constructs and initializes a QuadCurve from the specified coordinates.

Method Index

o **getBounds2D**()
    Return the bounding box of the shape.
o **getCtrlX**()
    Returns the X coordinate of the control point in double precision.
o **getCtrlY**()
    Returns the Y coordinate of the control point in double precision.
o **getX1**()
    Returns the X coordinate of the start point in double precision.
o **getX2**()
    Returns the X coordinate of the end point in double precision.
o **getY1**()
    Returns the Y coordinate of the start point in double precision.
o **getY2**()
    Returns the Y coordinate of the end point in double precision.
o **setCurve**(double, double, double, double, double, double)
    Sets the location of the endpoints and controlpoint of this curve to the specified double coordinates.

Variables

o **x1**

```
public float x1
```

The X coordinate of the start point of the quadratic curve segment.

o **y1**

```
public float y1
```

The Y coordinate of the start point of the quadratic curve segment.

o **ctrlx**

```
public float ctrlx
```

The X coordinate of the control point of the quadratic curve segment.

o **ctrly**

```
public float ctrly
```

The Y coordinate of the control point of the quadratic curve segment.

o **x2**

```
public float x2
```

The X coordinate of the end point of the quadratic curve segment.

o **y2**

```
public float y2
```

The Y coordinate of the end point of the quadratic curve segment.

Constructors

o **QuadCurve2D.Float**

```
public QuadCurve2D.Float()
```

Constructs and initializes a QuadCurve with coordinates (0, 0, 0, 0, 0, 0).

o **QuadCurve2D.Float**

```
public QuadCurve2D.Float(float x1,
                         float y1,
                         float ctrlx,
                         float ctrly,
                         float x2,
                         float y2)
```

Constructs and initializes a QuadCurve from the specified coordinates.

Methods

o **getX1**

```
public double getX1()
```

Returns the X coordinate of the start point in double precision.

Overrides:
getX1 in class QuadCurve2D

o **getY1**

```
public double getY1()
```

Returns the Y coordinate of the start point in double precision.

Overrides:
getY1 in class QuadCurve2D

o **getCtrlX**

```
public double getCtrlX()
```

Returns the X coordinate of the control point in double precision.

Overrides:
getCtrlX in class QuadCurve2D

o **getCtrlY**

```
public double getCtrlY()
```

Returns the Y coordinate of the control point in double precision.

Overrides:
getCtrlY in class QuadCurve2D

o **getX2**

```
public double getX2()
```

Returns the X coordinate of the end point in double precision.

Overrides:
getX2 in class QuadCurve2D

o **getY2**

```
public double getY2()
```

Returns the Y coordinate of the end point in double precision.

Overrides:
getY2 in class QuadCurve2D

o **setCurve**

```
public void setCurve(double x1,
                     double y1,
                     double ctrlx,
                     double ctrly,
                     double x2,
                     double y2)
```

Sets the location of the endpoints and controlpoint of this curve to the specified double coordinates.

Overrides:

setCurve in class QuadCurve2D

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the bounding box of the shape.

Class java.awt.geom.Rectangle2D

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Rectangle2D
```

public abstract class **Rectangle2D**
extends RectangularShape
super class of:
Rectangle, Rectangle2D.Double, Rectangle2D.Float

A rectangle defined by a location (x, y) and dimension (w x h).

This class is only the abstract superclass for all objects which store a 2D rectangle. The actual storage representation of the coordinates is left to the subclass.

Class Index

- **Rectangle2D.Double**
  A rectangle specified in double coordinates.
- **Rectangle2D.Float**
  A rectangle specified in float coordinates.

Variable Index

- **OUT_BOTTOM**
  The bitmask which indicates that a point lies below this Rectangle.
- **OUT_LEFT**
  The bitmask which indicates that a point lies to the left of this Rectangle.
- **OUT_RIGHT**
  The bitmask which indicates that a point lies to the right of this Rectangle.
- **OUT_TOP**
  The bitmask which indicates that a point lies above this Rectangle.

Constructor Index

o **Rectangle2D**()

Method Index

o **contains**(double, double)
 Test if a given coordinate is inside the boundary of the shape.
o **contains**(double, double, double, double)
 Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
o **getBounds2D**()
 Return the high precision bounding box of the shape.
o **getPathIterator**(AffineTransform)
 Return an iteration object that defines the boundary of the shape.
o **getPathIterator**(AffineTransform, double)
 Return an iteration object that defines the boundary of the flattened shape.
o **intersects**(double, double, double, double)
 Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.
o **intersectsLine**(double, double, double, double)
 Tests if the given line segment intersects the interior of this Rectangle.
o **intersectsLine**(Line2D)
 Tests if the given line segment intersects the interior of this Rectangle.
o **outcode**(double, double)
 Determines where the specified coordinates lie with respect to this Rectangle.
o **outcode**(Point2D)
 Determines where the specified point lies with respect to this Rectangle.
o **setBounds**(double, double, double, double)
 Sets the location and size of the outer bounds of this shape to the specified rectangular values.
o **setRect**(double, double, double, double)
 Sets the location and size of this rectangle to the specified double values.
o **setRect**(Rectangle2D)
 Sets this rectangle to be the same as the specified Rectangle.

Variables

o **OUT_LEFT**

```
public static final int OUT_LEFT
```

The bitmask which indicates that a point lies to the left of this Rectangle.

o **OUT_TOP**

```
public static final int OUT_TOP
```

The bitmask which indicates that a point lies above this Rectangle.

o **OUT_RIGHT**

```
public static final int OUT_RIGHT
```

The bitmask which indicates that a point lies to the right of this Rectangle.

o **OUT_BOTTOM**

```
public static final int OUT_BOTTOM
```

The bitmask which indicates that a point lies below this Rectangle.

Constructors

o **Rectangle2D**

```
protected Rectangle2D()
```

Methods

o **setRect**

```
public abstract void setRect(double x,
                             double y,
                             double w,
                             double h)
```

Sets the location and size of this rectangle to the specified double values.

o **setRect**

```
public void setRect(Rectangle2D r)
```

Sets this rectangle to be the same as the specified Rectangle.

o **intersectsLine**

```
public boolean intersectsLine(double x1,
                              double y1,
                              double x2,
                              double y2)
```

Tests if the given line segment intersects the interior of this Rectangle.

o **intersectsLine**

```
public boolean intersectsLine(Line2D l)
```

Tests if the given line segment intersects the interior of this Rectangle.

o **outcode**

```
public abstract int outcode(double x,
                            double y)
```

Determines where the specified coordinates lie with respect to this Rectangle. This method computes a binary OR of the appropriate mask values indicating which sides of the rectangle the given point is outside of.

Returns:
the logical OR of all appropriate out codes
See Also:
OUT_LEFT, OUT_TOP, OUT_RIGHT, OUT_BOTTOM

o **outcode**

```
public int outcode(Point2D p)
```

Determines where the specified point lies with respect to this Rectangle. This method computes a binary OR of the appropriate mask values indicating which sides of the rectangle the given point is outside of.

Returns:
the logical OR of all appropriate out codes
See Also:
OUT_LEFT, OUT_TOP, OUT_RIGHT, OUT_BOTTOM

o **setBounds**

```
public void setBounds(double x,
                      double y,
                      double w,
                      double h)
```

Sets the location and size of the outer bounds of this shape to the specified rectangular values.

Overrides:
setBounds in class RectangularShape

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
                                    double flatness)
```

Return an iteration object that defines the boundary of the flattened shape.

Overrides:
getPathIterator in class RectangularShape

Class java.awt.geom.Rectangle2D.Double

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Rectangle2D
                   |
                   +----java.awt.geom.Rectangle2D.Double
```

public static class **Rectangle2D.Double**
extends Rectangle2D

A rectangle specified in double coordinates.

Variable Index

o **height**
  The height of the rectangle.
o **width**
  The width of the rectangle.
o **x**
  The x coordinate of the rectangle.
o **y**
  The y coordinate of the rectangle.

Constructor Index

o **Rectangle2D.Double**()
  Constructs a new rectangle, initialized to location (0, 0) and size (0, 0).
o **Rectangle2D.Double**(double, double, double, double)
  Constructs and initializes a rectangle from the specified coordinates.

Method Index

o **getHeight**()
Returns the height of the rectangle in double precision.
o **getWidth**()
Returns the width of the rectangle in double precision.
o **getX**()
Returns the X coordinate of the rectangle in double precision.
o **getY**()
Returns the Y coordinate of the rectangle in double precision.
o **isEmpty**()
Determines whether the rectangle is empty.
o **outcode**(double, double)
Determines where the specified double coordinates lie with respect to this Rectangle.
o **setRect**(double, double, double, double)
Sets the location and size of this rectangle to the specified double values.
o **setRect**(Rectangle2D)
Sets this rectangle to be the same as the specified Rectangle.
o **toString**()
Returns the String representation of this Rectangle.

Variables

o **x**

```
public double x
```

The x coordinate of the rectangle.

o **y**

```
public double y
```

The y coordinate of the rectangle.

o **width**

```
public double width
```

The width of the rectangle.

o **height**

```
public double height
```

The height of the rectangle.

Constructors

o **Rectangle2D.Double**

```
public Rectangle2D.Double()
```

Constructs a new rectangle, initialized to location (0, 0) and size (0, 0).

o **Rectangle2D.Double**

```
public Rectangle2D.Double(double x,
                          double y,
                          double w,
                          double h)
```

Constructs and initializes a rectangle from the specified coordinates.

Parameters:
x – the x coordinate
y – the y coordinate
width – the width of the rectangle
height – the height of the rectangle

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the rectangle in double precision.

Overrides:
getX in class RectangularShape

o **getY**

```
public double getY()
```

Returns the Y coordinate of the rectangle in double precision.

Overrides:
getY in class RectangularShape

o **getWidth**

```
public double getWidth()
```

Returns the width of the rectangle in double precision.

Overrides:
getWidth in class RectangularShape

o **getHeight**

```
public double getHeight()
```

Returns the height of the rectangle in double precision.

Overrides:
getHeight in class RectangularShape

o **isEmpty**

```
public boolean isEmpty()
```

Determines whether the rectangle is empty.

Overrides:
isEmpty in class RectangularShape

o **setRect**

```
public void setRect(double x,
                    double y,
                    double w,
                    double h)
```

Sets the location and size of this rectangle to the specified double values.

Overrides:
setRect in class Rectangle2D

o **setRect**

```
public void setRect(Rectangle2D r)
```

Sets this rectangle to be the same as the specified Rectangle.

Overrides:
setRect in class Rectangle2D

o **outcode**

```
public int outcode(double x,
                   double y)
```

Determines where the specified double coordinates lie with respect to this Rectangle. This method computes a binary OR of the appropriate mask values indicating which sides of the rectangle the given point is outside of.

Returns:
the logical OR of all appropriate out codes

Overrides:
outcode in class Rectangle2D

See Also:
OUT_LEFT, OUT_TOP, OUT_RIGHT, OUT_BOTTOM

o **toString**

```
public String toString()
```

Returns the String representation of this Rectangle.

Overrides:
toString in class Object

Class java.awt.geom.Rectangle2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.Rectangle2D
                   |
                   +----java.awt.geom.Rectangle2D.Float
```

public static class **Rectangle2D.Float**
extends Rectangle2D

A rectangle specified in float coordinates.

Variable Index

o **height**
   The height of the rectangle.
o **width**
   The width of the rectangle.
o **x**
   The x coordinate of the rectangle.
o **y**
   The y coordinate of the rectangle.

Constructor Index

o **Rectangle2D.Float**()
   Constructs a new rectangle, initialized to location (0.0, 0.0) and size (0.0, 0.0).
o **Rectangle2D.Float**(float, float, float, float)
   Constructs and initializes a rectangle from the specified coordinates.

Method Index

o **getHeight**()
    Returns the height of the rectangle in double precision.
o **getWidth**()
    Returns the width of the rectangle in double precision.
o **getX**()
    Returns the X coordinate of the rectangle in double precision.
o **getY**()
    Returns the Y coordinate of the rectangle in double precision.
o **isEmpty**()
    Determines whether the rectangle is empty.
o **outcode**(double, double)
    Determines where the specified float coordinates lie with respect to this Rectangle.
o **setRect**(double, double, double, double)
    Sets the location and size of this rectangle to the specified double values.
o **setRect**(float, float, float, float)
    Sets the location and size of this rectangle to the specified float values.
o **setRect**(Rectangle2D)
    Sets this rectangle to be the same as the specified Rectangle.
o **toString**()
    Returns the String representation of this Rectangle.

Variables

o **x**

```
public float x
```

The x coordinate of the rectangle.

o **y**

```
public float y
```

The y coordinate of the rectangle.

o **width**

```
public float width
```

The width of the rectangle.

o **height**

```
public float height
```

The height of the rectangle.

Constructors

o **Rectangle2D.Float**

```
public Rectangle2D.Float()
```

Constructs a new rectangle, initialized to location (0.0, 0.0) and size (0.0, 0.0).

o **Rectangle2D.Float**

```
public Rectangle2D.Float(float x,
                         float y,
                         float w,
                         float h)
```

Constructs and initializes a rectangle from the specified coordinates.

Parameters:
x – the x coordinate
y – the y coordinate
width – the width of the rectangle
height – the height of the rectangle

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the rectangle in double precision.

Overrides:
getX in class RectangularShape

o **getY**

```
public double getY()
```

Returns the Y coordinate of the rectangle in double precision.

Overrides:
getY in class RectangularShape

o **getWidth**

```
public double getWidth()
```

Returns the width of the rectangle in double precision.

Overrides:
getWidth in class RectangularShape

o **getHeight**

```
public double getHeight()
```

Returns the height of the rectangle in double precision.

Overrides:
getHeight in class RectangularShape

o **isEmpty**

```
public boolean isEmpty()
```

Determines whether the rectangle is empty.

Overrides:
isEmpty in class RectangularShape

o **setRect**

```
public void setRect(float x,
                    float y,
                    float w,
                    float h)
```

Sets the location and size of this rectangle to the specified float values.

o **setRect**

```
public void setRect(double x,
                    double y,
                    double w,
                    double h)
```

Sets the location and size of this rectangle to the specified double values.

Overrides:
setRect in class Rectangle2D

o **setRect**

```
public void setRect(Rectangle2D r)
```

Sets this rectangle to be the same as the specified Rectangle.

Overrides:

setRect in class Rectangle2D

o **outcode**

```
public int outcode(double x,
                   double y)
```

Determines where the specified float coordinates lie with respect to this Rectangle. This method computes a binary OR of the appropriate mask values indicating which sides of the rectangle the given point is outside of.

Returns:
the logical OR of all appropriate out codes
Overrides:
outcode in class Rectangle2D
See Also:
OUT_LEFT, OUT_TOP, OUT_RIGHT, OUT_BOTTOM

o **toString**

```
public String toString()
```

Returns the String representation of this Rectangle.

Overrides:
toString in class Object

Class java.awt.geom.RectangularShape

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
```

---

public abstract class **RectangularShape**
extends Object
implements Shape
super class of:
Arc2D, Ellipse2D, Rectangle2D, RoundRectangle2D

The base class for a number of shapes which inscribe a rectangular set of outer bounds. This class provides common manipulation routines for operating on a shape by querying and modifying its bounding rectangle.

Constructor Index

o **RectangularShape**()

Method Index

o **contains**(Point2D)
Test if a given Point is inside the boundary of the shape.
o **contains**(Rectangle2D)
Test if the interior of the Shape entirely contains the given Rectangle.
o **getBottom**()
Returns the Y coordinate of the bottom edge of the shape in double precision.
o **getBounds**()
Return the bounding box of the shape.
o **getCenterX**()
Returns the X coordinate of the center of the shape in double precision.
o **getCenterY**()
Returns the Y coordinate of the center of the shape in double precision.
o **getHeight**()
Returns the height of the bounding rectangle in double precision.
o **getLeft**()

Returns the X coordinate of the left edge of the shape in double precision.

o **getPathIterator**(AffineTransform, double)
Return an iterator object that iterates along the boundary of the shape and provides access to a flattened view of the geometry of the outline of the shape.

o **getRight**()
Returns the X coordinate of the right edge of the shape in double precision.

o **getTop**()
Returns the Y coordinate of the top edge of the shape in double precision.

o **getWidth**()
Returns the width of the bounding rectangle in double precision.

o **getX**()
Returns the X coordinate of the bounding rectangle in double precision.

o **getY**()
Returns the Y coordinate of the bounding rectangle in double precision.

o **intersects**(Rectangle2D)
Test if the interior of the Shape intersects the interior of a given Rectangle.

o **isEmpty**()
Determines whether the rectangular shape is empty.

o **setBounds**(double, double, double, double)
Sets the location and size of the outer bounds of this shape to the specified rectangular values.

o **setBounds**(Point2D, Dimension2D)
Sets the outer bounds of this shape to be the same as the specified Point and Dimension objects.

o **setBounds**(Rectangle2D)
Sets the outer bounds of this shape to be the same as the specified Rectangle.

o **setBoundsFromCenter**(double, double, double, double)
Sets the outer bounds of this shape based on a center point and a corner point.

o **setBoundsFromCenter**(Point2D, Point2D)
Sets the outer bounds of this shape based on a center point and a corner point.

o **setBoundsFromDiagonal**(double, double, double, double)
Sets the outer bounds of this shape based on two points along one of its diagonals.

o **setBoundsFromDiagonal**(Point2D, Point2D)
Sets the outer bounds of this shape based on two Point objects along one of its diagonals.

Constructors

o **RectangularShape**

```
protected RectangularShape()
```

Methods

o **getX**

```
public abstract double getX()
```

Returns the X coordinate of the bounding rectangle in double precision.

o **getY**

```
public abstract double getY()
```

Returns the Y coordinate of the bounding rectangle in double precision.

o **getWidth**

```
public abstract double getWidth()
```

Returns the width of the bounding rectangle in double precision.

o **getHeight**

```
public abstract double getHeight()
```

Returns the height of the bounding rectangle in double precision.

o **getLeft**

```
public double getLeft()
```

Returns the X coordinate of the left edge of the shape in double precision.

o **getTop**

```
public double getTop()
```

Returns the Y coordinate of the top edge of the shape in double precision.

o **getRight**

```
public double getRight()
```

Returns the X coordinate of the right edge of the shape in double precision.

o **getBottom**

```
public double getBottom()
```

Returns the Y coordinate of the bottom edge of the shape in double precision.

o **getCenterX**

```
public double getCenterX()
```

Returns the X coordinate of the center of the shape in double precision.

o **getCenterY**

```
public double getCenterY()
```

Returns the Y coordinate of the center of the shape in double precision.

o **isEmpty**

```
public abstract boolean isEmpty()
```

Determines whether the rectangular shape is empty.

o **setBounds**

```
public abstract void setBounds(double x,
                               double y,
                               double w,
                               double h)
```

Sets the location and size of the outer bounds of this shape to the specified rectangular values.

o **setBounds**

```
public void setBounds(Point2D loc,
                      Dimension2D size)
```

Sets the outer bounds of this shape to be the same as the specified Point and Dimension objects.

o **setBounds**

```
public void setBounds(Rectangle2D r)
```

Sets the outer bounds of this shape to be the same as the specified Rectangle.

o **setBoundsFromDiagonal**

```
public void setBoundsFromDiagonal(double x1,
                                  double y1,
                                  double x2,
                                  double y2)
```

Sets the outer bounds of this shape based on two points along one of its diagonals.

o **setBoundsFromDiagonal**

```
public void setBoundsFromDiagonal(Point2D p1,
                                  Point2D p2)
```

Sets the outer bounds of this shape based on two Point objects along one of its diagonals.

o **setBoundsFromCenter**

```
public void setBoundsFromCenter(double centerX,
                                double centerY,
                                double cornerX,
                                double cornerY)
```

Sets the outer bounds of this shape based on a center point and a corner point.

o **setBoundsFromCenter**

```
public void setBoundsFromCenter(Point2D center,
                                Point2D corner)
```

Sets the outer bounds of this shape based on a center point and a corner point.

o **contains**

```
public boolean contains(Point2D p)
```

Test if a given Point is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(Rectangle2D r)
```

Test if the interior of the Shape intersects the interior of a given Rectangle.

o **contains**

```
public boolean contains(Rectangle2D r)
```

Test if the interior of the Shape entirely contains the given Rectangle.

o **getBounds**

```
public Rectangle getBounds()
```

Return the bounding box of the shape.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at,
                                    double flatness)
```

Return an iterator object that iterates along the boundary of the shape and provides access to a flattened view of the geometry of the outline of the shape. Only SEG_MOVETO, SEG_LINETO, and SEG_CLOSE point types will be returned by the iterator. The amount of subdivision of the curved segments is controlled by the `flatness` parameter which specifies ?REMIND?. An optional affine transform can be specified in which case the coordinates returned in the iteration will be transformed accordingly.

Parameters:

at – an optional AffineTransform to be applied to the coordinates as they are returned in the iteration, or null if the untransformed coordinates are desired.
flatness – the maximum amount that the control points for a given curve can vary from colinear before a subdivided curve is replaced by a straight line connecting the endpoints.

Class java.awt.geom.RoundRectangle2D

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.RoundRectangle2D
```

---

public abstract class **RoundRectangle2D**
extends RectangularShape
super class of:
RoundRectangle2D.Float

A rectangle with rounded corners defined by a location (x, y), a dimension (w x h), and the width and height of an arc to round the corners with.

This class is only the abstract superclass for all objects which store a 2D rounded rectangle. The actual storage representation of the coordinates is left to the subclass.

Class Index

o **RoundRectangle2D.Float**
A rectangle with rounded corners all specified in float coordinates.

Constructor Index

o **RoundRectangle2D**()

Method Index

o **contains**(double, double)
Test if a given coordinate is inside the boundary of the shape.
o **contains**(double, double, double, double)
Test if the interior of the Shape entirely contains the given set of rectangular coordinates.
o **getArcHeight**()
Gets the height of the arc that rounds off the corners.

o **getArcWidth**()
    Gets the width of the arc that rounds off the corners.
o **getPathIterator**(AffineTransform)
    Return an iteration object that defines the boundary of the shape.
o **intersects**(double, double, double, double)
    Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.
o **setBounds**(double, double, double, double)
    Sets the location and size of the outer bounds of this shape to the specified rectangular values.
o **setRoundRect**(double, double, double, double, double, double)
    Sets the location, size, and corner radii of this rounded rectangle to the specified double values.
o **setRoundRect**(RoundRectangle2D)
    Sets this rounded rectangle to be the same as the specified RoundRectangle.

Constructors

o **RoundRectangle2D**

```
protected RoundRectangle2D()
```

Methods

o **getArcWidth**

```
public abstract double getArcWidth()
```

Gets the width of the arc that rounds off the corners.

o **getArcHeight**

```
public abstract double getArcHeight()
```

Gets the height of the arc that rounds off the corners.

o **setRoundRect**

```
public abstract void setRoundRect(double x,
                                  double y,
                                  double w,
                                  double h,
                                  double arcWidth,
                                  double arcHeight)
```

Sets the location, size, and corner radii of this rounded rectangle to the specified double values.

o **setRoundRect**

```
public void setRoundRect(RoundRectangle2D rr)
```

Sets this rounded rectangle to be the same as the specified RoundRectangle.

o **setBounds**

```
public void setBounds(double x,
                      double y,
                      double w,
                      double h)
```

Sets the location and size of the outer bounds of this shape to the specified rectangular values.

Overrides:
setBounds in class RectangularShape

o **contains**

```
public boolean contains(double x,
                        double y)
```

Test if a given coordinate is inside the boundary of the shape.

o **intersects**

```
public boolean intersects(double x,
                          double y,
                          double w,
                          double h)
```

Test if the interior of the Shape intersects the interior of a given set of rectangular coordinates.

o **contains**

```
public boolean contains(double x,
                        double y,
                        double w,
                        double h)
```

Test if the interior of the Shape entirely contains the given set of rectangular coordinates.

o **getPathIterator**

```
public PathIterator getPathIterator(AffineTransform at)
```

Return an iteration object that defines the boundary of the shape.

Class java.awt.geom.RoundRectangle2D.Float

```
java.lang.Object
   |
   +----java.awt.geom.RectangularShape
           |
           +----java.awt.geom.RoundRectangle2D
                   |
                   +----java.awt.geom.RoundRectangle2D.Float
```

---

public static class **RoundRectangle2D.Float**
extends RoundRectangle2D

A rectangle with rounded corners all specified in float coordinates.

Variable Index

o **archeight**
The height of the arc that rounds off the corners.
o **arcwidth**
The width of the arc that rounds off the corners.
o **height**
The height of the rectangle.
o **width**
The width of the rectangle.
o **x**
The x coordinate of the rectangle.
o **y**
The y coordinate of the rectangle.

Constructor Index

o **RoundRectangle2D.Float**()
Constructs a new rounded rectangle, initialized to location (0.0, 0.0), size (0.0, 0.0), and corner arcs of radius 0.0.
o **RoundRectangle2D.Float**(float, float, float, float, float, float)
Constructs and initializes a rectangle from the specified coordinates.

Method Index

o **getArcHeight**()
    Gets the height of the arc that rounds off the corners.
o **getArcWidth**()
    Gets the width of the arc that rounds off the corners.
o **getBounds2D**()
    Return the high precision bounding box of the shape.
o **getHeight**()
    Returns the height of the rounded rectangle in double precision.
o **getWidth**()
    Returns the width of the rounded rectangle in double precision.
o **getX**()
    Returns the X coordinate of the rounded rectangle in double precision.
o **getY**()
    Returns the Y coordinate of the rounded rectangle in double precision.
o **isEmpty**()
    Determines whether the rounded rectangle is empty.
o **setRoundRect**(double, double, double, double, double, double)
    Sets the location, size, and arc radii of this rectangle to the specified double values.
o **setRoundRect**(float, float, float, float, float, float)
    Sets the location, size, and arc radii of this rectangle to the specified float values.
o **setRoundRect**(RoundRectangle2D)
    Sets this rounded rectangle to be the same as the specified RoundRectangle.

Variables

o **x**

```
public float x
```

The x coordinate of the rectangle.

o **y**

```
public float y
```

The y coordinate of the rectangle.

o **width**

```
public float width
```

The width of the rectangle.

o **height**

```
public float height
```

The height of the rectangle.

o **arcwidth**

```
public float arcwidth
```

The width of the arc that rounds off the corners.

o **archeight**

```
public float archeight
```

The height of the arc that rounds off the corners.

Constructors

o **RoundRectangle2D.Float**

```
public RoundRectangle2D.Float()
```

Constructs a new rounded rectangle, initialized to location (0.0, 0.0), size (0.0, 0.0), and corner arcs of radius 0.0.

o **RoundRectangle2D.Float**

```
public RoundRectangle2D.Float(float x,
                              float y,
                              float w,
                              float h,
                              float arcw,
                              float arch)
```

Constructs and initializes a rectangle from the specified coordinates.

Parameters:
x – the x coordinate
y – the y coordinate
width – the width of the rectangle
height – the height of the rectangle

Methods

o **getX**

```
public double getX()
```

Returns the X coordinate of the rounded rectangle in double precision.

Overrides:
getX in class RectangularShape

o **getY**

```
public double getY()
```

Returns the Y coordinate of the rounded rectangle in double precision.

Overrides:
getY in class RectangularShape

o **getWidth**

```
public double getWidth()
```

Returns the width of the rounded rectangle in double precision.

Overrides:
getWidth in class RectangularShape

o **getHeight**

```
public double getHeight()
```

Returns the height of the rounded rectangle in double precision.

Overrides:
getHeight in class RectangularShape

o **getArcWidth**

```
public double getArcWidth()
```

Gets the width of the arc that rounds off the corners.

Overrides:
getArcWidth in class RoundRectangle2D

o **getArcHeight**

```
public double getArcHeight()
```

Gets the height of the arc that rounds off the corners.

Overrides:

getArcHeight in class RoundRectangle2D

o **isEmpty**

```
public boolean isEmpty()
```

Determines whether the rounded rectangle is empty.

Overrides:
isEmpty in class RectangularShape

o **setRoundRect**

```
public void setRoundRect(float x,
                         float y,
                         float w,
                         float h,
                         float arcw,
                         float arch)
```

Sets the location, size, and arc radii of this rectangle to the specified float values.

o **setRoundRect**

```
public void setRoundRect(double x,
                         double y,
                         double w,
                         double h,
                         double arcw,
                         double arch)
```

Sets the location, size, and arc radii of this rectangle to the specified double values.

Overrides:
setRoundRect in class RoundRectangle2D

o **setRoundRect**

```
public void setRoundRect(RoundRectangle2D rr)
```

Sets this rounded rectangle to be the same as the specified RoundRectangle.

Overrides:
setRoundRect in class RoundRectangle2D

o **getBounds2D**

```
public Rectangle2D getBounds2D()
```

Return the high precision bounding box of the shape.

Class java.awt.geom.IncompatibleTransformException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----java.lang.RuntimeException
                           |
                           +----java.awt.geom.IncompatibleTransformException
```

---

public class **IncompatibleTransformException**
extends RuntimeException

This exception is thrown if an operation is attempted with incompatible Transform objects.

Constructor Index

o **IncompatibleTransformException**(String)

Constructors

o **IncompatibleTransformException**

```
 public IncompatibleTransformException(String s)
```

Class java.awt.geom.NoninvertibleTransformException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----java.awt.geom.NoninvertibleTransformException
```

public class **NoninvertibleTransformException**
extends Exception

This exception is thrown if an operation is performed requiring the inverse of a Transform object and the Transform is not invertible.

Constructor Index

o **NoninvertibleTransformException**(String)

Constructors

o **NoninvertibleTransformException**

```
public NoninvertibleTransformException(String s)
```

Interface java.awt.image.BufferedImageOp

public interface **BufferedImageOp**

This is an interface that describes single−input/single−output operations performed on BufferedImage objects. This is implemented by such classes as AffineTransformOp, ConvolveOp, BandCombineOp, and LookupOp. These objects can be passed into a BufferedImageFilter to operate on a BufferedImage in the ImageProducer−ImageFilter−ImageConsumer paradigm. This interface cannot be used to describe more sophisticated Ops such as ones that take multiple sources. Each class implementing this interface will specify whether or not it will allow an in−place filtering operation (i.e. source object equal to the destination object). Note that the restriction to single−input operations means that the values of destination pixels prior to the operation are not used as input to the filter operation.

See Also:
BufferedImage, BufferedImageFilter, AffineTransformOp, BandCombineOp, ColorConvertOp, ConvolveOp, LookupOp, RescaleOp, ThresholdOp

---

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
Creates an empty destination image with the correct size and number of components.
o **filter**(BufferedImage, BufferedImage)
Perform a single−input/single−output operation on a BufferedImage.
o **getDestBounds**(BufferedImage)
Returns the bounding box of the filtered destination image.
o **getDestPoint**(Point2D, Point2D)
Returns the location of the destination point given a point in the source image.

Methods

o **filter**

```
 public abstract BufferedImage filter(BufferedImage src,
                                      BufferedImage dest)
```

Perform a single−input/single−output operation on a BufferedImage. If the color models for the two images do not match, a color conversion into the destination color model will be performed. If the destination image is null, a BufferedImage with an appropriate ColorModel will be created. The IllegalArgumentException may be thrown if the source and/or destination image is incompatible with the types of images allowed by the subclass implementing this filter.

o **getDestBounds**

```
public abstract Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the filtered destination image. The IllegalArgumentException may be thrown if the source image is incompatible with the types of images allowed by the subclass implementing this filter.

o **createCompatibleDestImage**

```
public abstract BufferedImage createCompatibleDestImage(BufferedImage src,
                                                        ColorModel destCM)
```

Creates an empty destination image with the correct size and number of components. The IllegalArgumentException may be thrown if the source image is incompatible with the types of images allowed by the subclass implementing this filter.

Parameters:

src – Source image for the filter operation

destCM – ColorModel of the destination. If null, the ColorModel of the source will be used.

o **getDestPoint**

```
public abstract Point2D getDestPoint(Point2D srcPt,
                                     Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non−null, it will be used to hold the return value.

Interface java.awt.image.TileImageConsumer

public interface **TileImageConsumer**
extends ImageConsumer

The interface for objects expressing interest in image data through the ImageProducer interfaces. When a consumer is added to an image producer, the producer delivers all of the data about the image using the method calls defined in this interface.

See Also:
ImageProducer

Method Index

o **setPixels**(int, int, ColorModel, Tile)
The pixels of the image are delivered using one or more calls to the setPixels method.

Methods

o **setPixels**

```
 public abstract void setPixels(int x,
                                int y,
                                ColorModel model,
                                Tile tile)
```

The pixels of the image are delivered using one or more calls to the setPixels method. x and y specify the location and tile.getWidth() and tile.getHeight() specify the size of the rectangle of source pixels that are contained in the tile. The specified ColorModel object should be used to convert the pixels into their corresponding color and alpha components. Pixels in the tile can be accessed by using the getData method:

```
int[] pixel = tile.getData(tx, ty);
```

The variable pixel will contain one channel element for the specified location from each channel in the tile.

See Also:
ColorModel, Tile

Interface java.awt.image.TileOp

public interface **TileOp**

This is an interface that describes single–input/single–output operations performed on Tile objects. This is implemented by such classes as AffineTransformOp, ConvolveOp, and LookupOp. The Source and Destination objects must contain the appropriate number of channels for the particular classes implementing this interface. Otherwise, an exception is thrown. This interface cannot be used to describe more sophisticated Ops such as ones that take multiple sources. Each class implementing this interface will specify whether or not it will allow an in–place filtering operation (i.e. source object equal to the destination object). Note that the restriction to single–input operations means that the values of destination pixels prior to the operation are not used as input to the filter operation.

See Also:
AffineTransformOp, BandCombineOp, ColorConvertOp, ConvolveOp, LookupOp, RescaleOp, ThresholdOp

---

Method Index

o **createCompatibleDestTile**(Tile)
Creates an empty destination tile with the correct size and number of channels.
o **filter**(Tile, Tile)
Performs a single–input/single–output operation from a source Tile to a destination Tile.
o **getDestBounds**(Tile)
Returns the bounding box of the filtered destination Tile.
o **getDestPoint**(Point2D, Point2D)
Returns the location of the destination point given a point in the source tile.

Methods

o **filter**

```
public abstract Tile filter(Tile src,
                            Tile dest)
```

Performs a single–input/single–output operation from a source Tile to a destination Tile. If the destination Tile does not exist, a new tile will be created. The IllegalArgumentException may be thrown if the source and/or destination tile is incompatible with the types of tiles allowed by the subclass implementing this filter.

o **getDestBounds**

```
public abstract Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the filtered destination Tile. The IllegalArgumentException may be thrown if the source tile is incompatible with the types of tiles allowed by the subclass implementing this filter.

o **createCompatibleDestTile**

```
public abstract Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels. The IllegalArgumentException may be thrown if the source tile is incompatible with the types of tiles allowed by the subclass implementing this filter.

o **getDestPoint**

```
public abstract Point2D getDestPoint(Point2D srcPt,
                                     Point2D dstPt)
```

Returns the location of the destination point given a point in the source tile. If dstPt is non−null, it will be used to hold the return value.

Class java.awt.image.AffineTransformOp

```
java.lang.Object
   |
   +----java.awt.image.AffineTransformOp
```

public abstract class **AffineTransformOp**
extends Object
implements BufferedImageOp, TileOp
super class of:
    BilinearAffineTransformOp, NearestNeighborAffineTransformOp

This abstract class uses an affine transform to perform a linear mapping from 2D coordinates in the source image or tile to 2D coordinates in the destination image or tile. The subclasses define what kind of interpolation will be done when the image is transformed. These subclasses include: BilinearAffineTransformOp and NearestNeighborAffineTransformOp. This class of objects can be passed into a BufferedImageFilter to operate on a BufferedImage within the Image Producer/ Consumer model.

Note that the following constraints have to be met:

- Source and Destination must be different.
- For Tiles, the number of Channels in the Source must be equal to the number of Channels in the Destination.

See Also:
    AffineTransform, BilinearAffineTransformOp, NearestNeighborAffineTransformOp, BufferedImageFilter

Constructor Index

o **AffineTransformOp**(AffineTransform)
    Constructs an AffineTransformOp given an affine transform.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
    Creates an empty destination image with the correct size and number of components.
o **createCompatibleDestTile**(Tile)
    Creates an empty destination tile with the correct size and number of channels.
o **filter**(BufferedImage, BufferedImage)
    Abstract method to transform the BufferedImage.
o **filter**(Tile, Tile)
    Abstract method to transform the Tile.
o **getDestBounds**(BufferedImage)
    Returns the bounding box of the transformed destination.
o **getDestBounds**(Tile)
    Returns the bounding box of the transformed destination.
o **getDestPoint**(Point2D, Point2D)
    Returns the location of the destination point given a point in the source image.
o **getTransform**()
    Returns the current transform.

Constructors

o **AffineTransformOp**

```
protected AffineTransformOp(AffineTransform xform)
```

Constructs an AffineTransformOp given an affine transform.

Methods

o **filter**

```
public abstract BufferedImage filter(BufferedImage src,
                                     BufferedImage dest)
```

Abstract method to transform the BufferedImage. If the color models for the two images do not match, a color conversion into the destination color model will be performed. If the destination image is null, a BufferedImage will be created with the source ColorModel. The IllegalArgumentException may be thrown if the source is the same as the destination.

o **filter**

```
public abstract Tile filter(Tile src,
                            Tile dest)
```

Abstract method to transform the Tile. This operation will perform the transform channel by channel. If the destination Tile is null, a new Tile will be created. The IllegalArgumentException may be thrown if the source is the same as the destination or if the number of channels in the source is not equal to the number of channels in the destination.

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the transformed destination.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the transformed destination.

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of components.

Parameters:

src − Source image for the filter operation
destCM − ColorModel of the destination. If null, the ColorModel of the source will be used.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                            Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non−null, it will be used to hold the return value.

o **getTransform**

```
public AffineTransform getTransform()
```

Returns the current transform.

Class java.awt.image.BandCombineOp

```
java.lang.Object
   |
   +----java.awt.image.BandCombineOp
```

public class **BandCombineOp**
extends Object
implements BufferedImageOp, TileOp

This class performs an arbitrary linear combination of image components in an image or bands in a tile, using a specified matrix.

For operations with a BufferedImage, the width of the matrix must be equal to the number of components in the source, or this number plus or minus one. If the width of the matrix is equal to the number of components in the source plus one, then there is an implied 1 at the end of the pixel vector. For example, an RGB image without alpha might have the following transformation applied to each pixel in order to invert the second band of the image.

```
   [ 1.0   0.0   0.0    0.0  ]     [ r ]
   [ 0.0  -1.0   0.0  255.0  ]  x  [ g ]
   [ 0.0   0.0   0.0    0.0  ]     [ b ]
                                   [ 1 ]
```

If the width of the matrix is equal to the number of components in the source minus one, there must be an alpha component in the image and that component will be ignored in this operation. The height of the matrix must be equal to the number of components in the destination. Thus, the source and destination can have a different number of components. For instance, one could select the third component from an image with 4 components using the following matrix:

```
   [ 0.0  0.0  1.0  0.0 ]
```

For operations with a Tile, the width of the matrix must be equal to the number of bands, optionally plus one. If there is one more column in the matrix than the number of bands, there is an implied 1 at the end of the pixel vector. The height of the matrix must be equal to the number of bands in the destination.

Note that the source and destination can be the same object.

Constructor Index

o **BandCombineOp**(float[][])
    This constructs a BandCombineOp with the specified matrix.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
    Creates an empty destination image with the correct size and number of components.
o **createCompatibleDestTile**(Tile)
    Creates an empty destination tile with the correct size and number of bands.
o **filter**(BufferedImage, BufferedImage)
    Transforms the BufferedImage using the matrix in the constructor.
o **filter**(Tile, Tile)
    Transforms the Tile using the matrix in the constructor.
o **getDestBounds**(BufferedImage)
    Returns the bounding box of the transformed destination.
o **getDestBounds**(Tile)
    Returns the bounding box of the transformed destination.
o **getDestPoint**(Point2D, Point2D)
    Returns the location of the destination point given a point in the source image.
o **getMatrix**()
    Returns the matrix.

Constructors

o **BandCombineOp**

```
public BandCombineOp(float matrix[][])
```

This constructs a BandCombineOp with the specified matrix. See the class comments for restrictions on the size of the matrix. The first subscript is the row index and the second is the column index.

Methods

o **getMatrix**

```
public float[][] getMatrix()
```

Returns the matrix.

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst)
```

Transforms the BufferedImage using the matrix in the constructor. This method will throw an IllegalArgumentException if the number of components in the source or destination is incompatible with the matrix. See the class comments for more details. If the destination image is null, a BufferedImage will be created with the source ColorModel. This filtering operation will not color convert the pixels from the source to the destination color model.

o **filter**

```
public Tile filter(Tile src,
                   Tile dst)
```

Transforms the Tile using the matrix in the constructor. The IllegalArgumentException may be thrown if the number of bands in the source or destination is incompatible with the matrix. See the class comments for more details. If the destination is null, it will be created with a number of bands equalling the number of rows in the matrix.

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the transformed destination. Since this is not a geometric operation, the bounding box does not change. The IllegalArgumentException may be thrown if the number of components in the source is incompatible with the matrix. See the class comments for more details.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the transformed destination. Since this is not a geometric operation, the bounding box does not change. The IllegalArgumentException may be thrown if the number of bands in the source is incompatible with the matrix. See the class comments for more details.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of bands. The IllegalArgumentException may be thrown if the number of bands in the source is incompatible with the matrix. See the class comments for more details.

Parameters:
src – Source tile for the filter operation

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of components. The IllegalArgumentException may be thrown if the number of components in the source is incompatible with the matrix, or if the number of components in the destination color model does not equal the number of rows in the matrix. See the class comments for more details.

Parameters:

src − Source image for the filter operation
destCM − ColorModel of the destination. If null, the ColorModel of the source will be used.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                            Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non−null, it will be used to hold the return value. Since this is not a geometric operation, the srcPt will equal the dstPt.

Class java.awt.image.BilinearAffineTransformOp

```
java.lang.Object
   |
   +----java.awt.image.AffineTransformOp
           |
           +----java.awt.image.BilinearAffineTransformOp
```

public class **BilinearAffineTransformOp**
extends AffineTransformOp

This class uses an affine transformation with bilinear interpolation to transform an image or tile.

Note that the following constraints have to be met:

- Source and Destination must be different.
- For Tiles, the number of Channels in the Source must be equal to the number of Channels in the Destination.

Constructor Index

o **BilinearAffineTransformOp**(AffineTransform)
Constructs a BilinearAffineTransformOp given an affine transform.

Method Index

o **filter**(BufferedImage, BufferedImage)
Transforms the BufferedImage using the stored affine transform.
o **filter**(Tile, Tile)
Transforms the Tile using the stored affine transform.

Constructors

o **BilinearAffineTransformOp**

```
public BilinearAffineTransformOp(AffineTransform xform)
```

Constructs a BilinearAffineTransformOp given an affine transform.

Methods

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst)
```

Transforms the BufferedImage using the stored affine transform. If the color models for the two images do not match, a color conversion into the destination color model will be performed. If the destination image is null, a BufferedImage will be created with the source ColorModel. The IllegalArgumentException may be thrown if the source is the same as the destination.

Overrides:
filter in class AffineTransformOp

o **filter**

```
public Tile filter(Tile src,
                   Tile dest)
```

Transforms the Tile using the stored affine transform. This operation will perform the transform channel by channel. If the destination Tile is null, a new Tile will be created. The IllegalArgumentException may be thrown if the source is the same as the destination or if the number of channels in the source is not equal to the number of channels in the destination.

Overrides:
filter in class AffineTransformOp

Class java.awt.image.BufferedImage

```
java.lang.Object
   |
   +----java.awt.Image
           |
           +----java.awt.image.BufferedImage
```

---

public class **BufferedImage**
extends Image

This subclass describes an Image with an accessible buffer of image data. A BufferedImage is comprised of a ColorModel and a Tile of image data. The number and types of Channel objects in the Tile must match the number and types required by the ColorModel to represent its color and alpha components. See the class comments in the Tile class for further discussion of how to manipulate multi−channelled data.

See Also:
ColorModel, Tile

Variable Index

o **TYPE_3BYTE_BGR**
Represents an image with 8−bit RGB color components (corresponds to a Windows−style BGR color model) with the colors Blue, Green, and Red stored in 3 bytes.
o **TYPE_4BYTE_ABGR**
Represents an image with 8−bit RGBA color components with the colors Blue, Green, and Red stored in 3 bytes and 1 byte of alpha.
o **TYPE_4BYTE_ABGR_PRE**
Represents an image with 8−bit RGBA color components with the colors Blue, Green, and Red stored in 3 bytes and 1 byte of alpha.
o **TYPE_BINARY**
Represents an opaque binary image.
o **TYPE_CUSTOM**
Image type is not recognized so it must be a customized image.
o **TYPE_INT_ARGB**
Represents an image with 8−bit RGBA color components packed into an integer.
o **TYPE_INT_ARGB_PRE**

Represents an image with 8−bit RGBA color components packed into an integer.

o **TYPE_INT_RGB**

Represents an image with 8−bit RGB color components packed into an integer.

Constructor Index

o **BufferedImage**(ColorModel, Tile, boolean)

Constructs a new BufferedImage with a given ColorModel and Tile.

o **BufferedImage**(int, int, int)

Constructs a BufferedImage of one of the predefined image types.

o **BufferedImage**(int, int, int, IndexColorModel)

Constructs a BufferedImage of one of the predefined image types: TYPE_BINARY

Method Index

o **coerceData**(boolean)

Forces the data to match the state specified in the isAlphaPremultiplied variable.

o **createGraphics**()

Creates a Graphics2D, which can be used to draw into this BufferedImage.

o **flush**()

Flushes all resources being used to cache optimization information.

o **getAlphaTile**()

Returns the alpha channel for ColorModel objects that support a separate spatial alpha channel (such as ComponentColorModel, PackedColorModel, and some instances of IndexColorModel).

o **getColorModel**()

Returns the ColorModel.

o **getGraphics**()

This method will actually return a Graphics2D but is here for backwards compatibility. **Deprecated.**

o **getHeight**()

Returns the height of the BufferedImage.

o **getHeight**(ImageObserver)

Returns the actual height of the image.

o **getProperty**(String)

Returns a property of the image by name.

o **getProperty**(String, ImageObserver)

Returns a property of the image by name.

o **getRGB**(int, int)

Returns an integer pixel in the default RGB color model (TYPE_INT_ARGB) and default sRGB colorspace.

o **getRGBArray**(int, int, int, int, int[], int, int)

Returns an array of integer pixels in the default RGB color model (TYPE_INT_ARGB) and default sRGB color space, from a portion of the image data.

o **getSource**()

Returns the object that produces the pixels for the image.

o **getSubimage**(int, int, int, int)

Returns a subimage given a rectangular region.

o **getTile**()

Returns the Tile.

o **getType**()

Returns the image type.

o **getWidth**()

Returns the width of the BufferedImage.

o **getWidth**(ImageObserver)

Returns the actual width of the image.

o **isAlphaPremultiplied**()

Returns whether or not the alpha has been premultiplied.

o **setRGB**(int, int, int)

Set a pixel in a BufferedImage.

o **toString**()

Variables

o TYPE_CUSTOM

```
public static final int TYPE_CUSTOM
```

Image type is not recognized so it must be a customized image. This type is only used as a return value for the getType() method.

o TYPE_INT_RGB

```
public static final int TYPE_INT_RGB
```

Represents an image with 8−bit RGB color components packed into an integer. The image has an IntDiscreteChannel and a DirectColorModel (without alpha).

o TYPE_INT_ARGB

```
public static final int TYPE_INT_ARGB
```

Represents an image with 8−bit RGBA color components packed into an integer. The has an IntDiscreteChannel and a DirectColorModel (with alpha). The color data in this image is considered not to be premultiplied with alpha. When this type is used as the imageType argument to a BufferedImage constructor, the image created will be consistent with images created in the JDK1.1 and earlier releases.

o TYPE_INT_ARGB_PRE

```
public static final int TYPE_INT_ARGB_PRE
```

Represents an image with 8−bit RGBA color components packed into an integer. The image has an IntDiscreteChannel and a DirectColorModel (with alpha). The color data in this image is considered to be premultiplied with alpha.

o **TYPE_3BYTE_BGR**

```
public static final int TYPE_3BYTE_BGR
```

Represents an image with 8−bit RGB color components (corresponds to a Windows−style BGR color model) with the colors Blue, Green, and Red stored in 3 bytes. There is no alpha. The image has 3 ByteDiscreteChannel objects and a ComponentColorModel.

o **TYPE_4BYTE_ABGR**

```
public static final int TYPE_4BYTE_ABGR
```

Represents an image with 8−bit RGBA color components with the colors Blue, Green, and Red stored in 3 bytes and 1 byte of alpha. The image has 4 ByteDiscreteChannel objects and a ComponentColorModel (with alpha). The color data in this image will be considered not to be premultiplied with alpha. The byte data is interleaved in a single byte array in the order A, B, G, R from lower to higher byte addresses within each pixel.

o **TYPE_4BYTE_ABGR_PRE**

```
public static final int TYPE_4BYTE_ABGR_PRE
```

Represents an image with 8−bit RGBA color components with the colors Blue, Green, and Red stored in 3 bytes and 1 byte of alpha. The image has 4 ByteDiscreteChannel objects and a ComponentColorModel (with alpha). The color data in this image will be considered to be premultiplied with alpha. The byte data is interleaved in a single byte array in the order A, B, G, R from lower to higher byte addresses within each pixel.

o **TYPE_BINARY**

```
public static final int TYPE_BINARY
```

Represents an opaque binary image. The image has one BytePackedChannel object and an IndexColorModel (without alpha). When this type is used as the imageType argument to the BufferedImage constructor which takes an imageType argument but no ColorModel argument, an IndexColorModel will be created with two colors in the default sRGB ColorSpace: {0, 0, 0} and {255, 255, 255}.

Constructors

o **BufferedImage**

```
public BufferedImage(int width,
                     int height,
                     int imageType)
```

Constructs a BufferedImage of one of the predefined image types. The ColorSpace for the image will be the default sRGB space.

Parameters:
: width – Width of the created image.
height – Height of the created image.
imageType – Type of the created image.

See Also:
: ColorSpace, TYPE_INT_RGB, TYPE_INT_ARGB, TYPE_INT_ARGB_PRE, TYPE_3BYTE_BGR, TYPE_4BYTE_ABGR, TYPE_4BYTE_ABGR_PRE, TYPE_BINARY

o **BufferedImage**

```
public BufferedImage(int width,
                     int height,
                     int imageType,
                     IndexColorModel cm)
```

Constructs a BufferedImage of one of the predefined image types: TYPE_BINARY

Parameters:
: width – Width of the created image.
height – Height of the created image.
imageType – Type of the created image.
cm – IndexColorModel of the created image.

See Also:
: TYPE_BINARY

o **BufferedImage**

```
public BufferedImage(ColorModel cm,
                     Tile tile,
                     boolean isTilePremultiplied)
```

Constructs a new BufferedImage with a given ColorModel and Tile. If the number and types of channels in the Tile do not match the requirements of the ColorModel for representing its color and alpha components, an exception will be thrown. This method may multiply or divide the color Tile data by alpha to match the alphaPremultiplied state in the ColorModel.

Parameters:
: ColorModel – ColorModel for the new image
tile – Tile for the data buffer
isTilePremultiplied – If true, the data in the tile has been premultiplied with alpha.

See Also:
: ColorModel, Tile

Methods

o **getType**

```
public int getType()
```

Returns the image type. If it is not one of the known types, TYPE_CUSTOM is returned.

See Also:

TYPE_INT_RGB, TYPE_INT_ARGB, TYPE_INT_ARGB_PRE, TYPE_3BYTE_BGR, TYPE_4BYTE_ABGR, TYPE_4BYTE_ABGR_PRE, TYPE_BINARY, TYPE_CUSTOM

o **getColorModel**

```
public ColorModel getColorModel()
```

Returns the ColorModel.

o **getTile**

```
public Tile getTile()
```

Returns the Tile.

o **getAlphaTile**

```
public Tile getAlphaTile()
```

Returns the alpha channel for ColorModel objects that support a separate spatial alpha channel (such as ComponentColorModel, PackedColorModel, and some instances of IndexColorModel). This may return null if there is no alpha channel associated with ColorModel in this image. If the image uses an IndexColorModel which has alpha in the lookup table, this method will return null since there is no spatially discrete alpha channel. This method will create a new channel (but will share the data array) if the image uses a DirectColorModel that has alpha since the alpha would be implicitly packed into an integer.

o **getRGB**

```
public int getRGB(int x,
                  int y)
```

Returns an integer pixel in the default RGB color model (TYPE_INT_ARGB) and default sRGB colorspace. Color conversion will take place if this default model does not match the image ColorModel. There are only 8−bits of precision for each color component in the returned data when using this method.

o **getRGBArray**

```
public int[] getRGBArray(int startX,
                         int startY,
                         int w,
                         int h,
                         int rgbArray[],
```

```
                              int offset,
                              int scansize)
```

Returns an array of integer pixels in the default RGB color model (TYPE_INT_ARGB) and default sRGB color space, from a portion of the image data. Color conversion will take place if the default model does not match the image ColorModel. There are only 8−bits of precision for each color component in the returned data when using this method. Given a coordinate (x,y) in the image, the ARGB pixel can be accessed using the following:

```
    pixel   = rgbArray[offset + ((y-startY)*w + (x-startX))*scansize];
```

Parameters:
startX − Starting x Coordinate
startY − Starting y Coordinate
w − Width of region
h − Height of region
rgbArray − If non−null, the rgb pixels will be written here
offset − Offset into the rgbArray
scansize − Stride for the rgbArray

Returns:
Array of RGB pixels.

o **setRGB**

```
public synchronized void setRGB(int x,
                                int y,
                                int rgb)
```

Set a pixel in a BufferedImage. The pixel is assumed to be in the default RGB color model (TYPE_INT_ARGB) and default sRGB color space. For images with an IndexColorModel, the index with the nearest color will be chosen.

o **getWidth**

```
public int getWidth()
```

Returns the width of the BufferedImage.

o **getHeight**

```
public int getHeight()
```

Returns the height of the BufferedImage.

o **getWidth**

```
public int getWidth(ImageObserver observer)
```

Returns the actual width of the image. If the width is not known yet, then the ImageObserver will be notified later and −1 will be returned.

Overrides:
getWidth in class Image
See Also:
getHeight, ImageObserver

o **getHeight**

```
public int getHeight(ImageObserver observer)
```

Returns the actual height of the image. If the height is not known yet, then the ImageObserver will be notified later and −1 will be returned.

Overrides:
getHeight in class Image
See Also:
getWidth, ImageObserver

o **getSource**

```
public ImageProducer getSource()
```

Returns the object that produces the pixels for the image. This may return null.

Overrides:
getSource in class Image
See Also:
ImageProducer

o **getProperty**

```
public Object getProperty(String name,
                          ImageObserver observer)
```

Returns a property of the image by name. Individual property names are defined by the various image formats. If a property is not defined for a particular image, this method will return the UndefinedProperty object. If the properties for this image are not yet known, then this method will return null and the ImageObserver object will be notified later. The property name "comment" should be used to store an optional comment which can be presented to the user as a description of the image, its source, or its author.

Overrides:
getProperty in class Image
See Also:
ImageObserver, UndefinedProperty

o **getProperty**

```
public Object getProperty(String name)
```

Returns a property of the image by name.

o **flush**

```
public void flush()
```

Flushes all resources being used to cache optimization information. The underlying pixel data is unaffected.

Overrides:
flush in class Image

o **getGraphics**

```
public Graphics getGraphics()
```

Note: getGraphics() is deprecated.

This method will actually return a Graphics2D but is here for backwards compatibility.

Overrides:
getGraphics in class Image

o **createGraphics**

```
public Graphics2D createGraphics()
```

Creates a Graphics2D, which can be used to draw into this BufferedImage.

o **getSubimage**

```
public BufferedImage getSubimage(int x,
                                 int y,
                                 int w,
                                 int h)
```

Returns a subimage given a rectangular region. The returned BufferedImage will share the same data array as the original image.

o **isAlphaPremultiplied**

```
public boolean isAlphaPremultiplied()
```

Returns whether or not the alpha has been premultiplied. It will return true if there is no alpha since the default alpha is OPAQUE.

o **coerceData**

```
public void coerceData(boolean isAlphaPremultiplied)
```

Forces the data to match the state specified in the isAlphaPremultiplied variable. It may multiply or divide the color Tile data by alpha, or do nothing if the data is in the correct state.

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

Class java.awt.image.BufferedImageFilter

```
java.lang.Object
   |
   +----java.awt.image.ImageFilter
           |
           +----java.awt.image.BufferedImageFilter
```

---

public class **BufferedImageFilter**
extends ImageFilter
implements TileImageConsumer, Cloneable

This class subclasses an ImageFilter to provide a simple means of using a single−source/single−destination image operator (BufferedImageOp) to filter a BufferedImage or Tile in the Image Producer/Consumer/Observer paradigm. Examples of these image operators are: ConvolveOp, AffineTransformOp and LookupOp.

See Also:
ImageFilter, BufferedImage, Tile, BufferedImageOp

Constructor Index

o **BufferedImageFilter**(BufferedImageOp)
Constructs a BufferedImageFilter with the specified single−source/single−destination operator.

Method Index

o **getBufferedImageOp**()
Returns the BufferedImageOp.
o **getFilterInstance**(ImageConsumer)
Returns a unique instance of a BufferedImageFilter object which will actually perform the filtering for the specified ImageConsumer.
o **setPixels**(int, int, ColorModel, Tile)
Filters the information provided in the setPixels method of the TileImageConsumer interface which takes a Tile.

Constructors

o **BufferedImageFilter**

```
public BufferedImageFilter(BufferedImageOp op)
```

Constructs a BufferedImageFilter with the specified single−source/single−destination operator.

Methods

o **getBufferedImageOp**

```
public BufferedImageOp getBufferedImageOp()
```

Returns the BufferedImageOp.

o **getFilterInstance**

```
public ImageFilter getFilterInstance(ImageConsumer ic)
```

Returns a unique instance of a BufferedImageFilter object which will actually perform the filtering for the specified ImageConsumer.

Overrides:
getFilterInstance in class ImageFilter

o **setPixels**

```
public void setPixels(int x,
                      int y,
                      ColorModel model,
                      Tile tile)
```

Filters the information provided in the setPixels method of the TileImageConsumer interface which takes a Tile. The filter used will be the BufferedImageOp that was passed to the constructor.

See Also:
setPixels

Class java.awt.image.ByteBandedTile

```
java.lang.Object
   |
   +----java.awt.image.Tile
           |
           +----java.awt.image.ByteBandedTile
```

public class **ByteBandedTile**
extends Tile

This class defines a tile with pixels consisting of multiple 8−bit component stored in separate arrays for each component. Operations on sets of pixels are performed on a given component of each pixel in the set before moving on to the next component. The arrays used for storage may be distinct or shared between some or all of the components. There is only one pixel stride and one scanline stride for all channels. This type of tile can be used with a ComponentColorModel.

The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Constructor Index

o **ByteBandedTile**(int, int, int, int, int, int[], int, int, byte[][])
    Constructs a base tile given the tile parameters.
o **ByteBandedTile**(int, int, int, int[], int, int, byte[][])
    Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
    Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.
o **createCompatibleTile**(int, int)
    Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.

- o **createSubtile**(int, int, int, int)
  Creates a subtile given a region of the tile.
- o **createSubtile**(int, int, int, int, int[])
  Creates a subtile given a region of the tile.
- o **getByteData**(int, int, int, int, byte[])
  Returns a byte array of pixels from the specified rectangular region.
- o **getByteData**(int, int, int, int, int, byte[])
  Returns a byte array of pixels from the specified rectangular region.
- o **getChannelOffset**(int)
  Returns channel offset for the specified channel.
- o **getChannelOffsets**()
  Returns a copy of the channel offsets array.
- o **getData**(int, int, int)
  Returns the channel element for the specified channel at the specified location.
- o **getData**(int, int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.
- o **getData**(int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.
- o **getData**(int, int, int[])
  Returns the channel elements for all channels at the specified location.
- o **getDataStorage**()
  Returns a reference to the entire data array
- o **getDataStorage**(int)
  Returns a reference to the specific channel data array
- o **getPixelStride**()
  Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.
- o **getScanlineStride**()
  Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.
- o **putByteData**(int, int, int, int, byte[])
  Stores a byte array of pixels into the specified rectangular region.
- o **putByteData**(int, int, int, int, int, byte[])
  Stores a byte array of pixels into the specified rectangular region.
- o **putData**(int, int, int, int)
  Stores the channel element for the specified channel at the specified location.
- o **putData**(int, int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.
- o **putData**(int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.
- o **putData**(int, int, int[])
  Stores the channel elements for all channels at the specified location.
- o **putData**(int, int, Tile)
  Stores the Tile data at the specified location.
- o **toString**()

Constructors

o **ByteBandedTile**

```
public ByteBandedTile(int width,
                      int height,
                      int numChannels,
                      int channelOffsets[],
                      int pixelStride,
                      int scanlineStride,
                      byte data[][])
```

Constructs a base tile given the channel array and tile dimensions. The tile data will start at x=0, y=0 in the data array of each channel.

Parameters:
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile width and height.

o **ByteBandedTile**

```
public ByteBandedTile(int x,
                      int y,
                      int width,
                      int height,
                      int numChannels,
                      int channelOffsets[],
                      int pixelStride,
                      int scanlineStride,
                      byte data[][])
```

Constructs a base tile given the tile parameters.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffset – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.

pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException

if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getChannelOffsets**

```
public int[] getChannelOffsets()
```

Returns a copy of the channel offsets array.

o **getChannelOffset**

```
public int getChannelOffset(int channel)
```

Returns channel offset for the specified channel. The channel offset is the index into the channel data array in which the first channel element of the first scanline is stored.

o **getScanlineStride**

```
public int getScanlineStride()
```

Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **getPixelStride**

```
public int getPixelStride()
```

Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.

o **getDataStorage**

```
public byte[][] getDataStorage()
```

Returns a reference to the entire data array

o **getDataStorage**

```
public byte[] getDataStorage(int channel)
```

Returns a reference to the specific channel data array

o **getData**

```
public int getData(int x,
                   int y,
                   int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int outData[])
```

Returns the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
outData – If non–null, channel elements for all channels at the specified location are returned in this array.
Returns:
Data array with channel elements for all channels.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int channel,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown

at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
int[] pixel = new int[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

Overrides:

getData in class Tile

o **getByteData**

```
public byte[] getByteData(int x,
                          int y,
                          int w,
                          int h,
                          int channel,
                          byte outData[])
```

Returns a byte array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
byte[] channelData = tile.getByteData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
byte chanElement = channelData[((y2-y)*w + (x2-x))];
```

o **getByteData**

```
public byte[] getByteData(int x,
                          int y,
                          int w,
                          int h,
                          byte outData[])
```

Returns a byte array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
byte[] channelData = tile.getByteData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
byte[] pixel = new byte[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

o **putData**

```
public void putData(int x,
                    int y,
                    int channel,
                    int inData)
```

Stores the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
inData – Channel element.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inData – Channel elements for all channels.
Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inTile – Tile of data to place at x,y location.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int channel,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are

assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putByteData**

```
public void putByteData(int x,
                        int y,
                        int w,
                        int h,
                        int channel,
                        byte inData[])
```

Stores a byte array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **putByteData**

```
public void putByteData(int x,
                        int y,
                        int w,
                        int h,
                        byte inData[])
```

Stores a byte array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width of the subtile.
height − Height of the subtile.

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height,
                          int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. A subset of the parent tile's channels may be specified. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width (in pixels) of the tile
height − Height (in pixels) of the tile
channelList − Array of channel indices.

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Parameters that are guaranteed to remain the same are:

- number of channels
- pixel stride

Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
```

```
         may not be equal to:
gc.createCompatibleTile(w2, h2)
```

Overrides:

createCompatibleTile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays. If the tile is a subtile, this will call createCompatibleTile(width, height).

Overrides:

createCompatibleTile in class Tile

o **toString**

```
public String toString()
```

Overrides:

toString in class Object

Class java.awt.image.ByteComponentTile

```
java.lang.Object
   |
   +----java.awt.image.Tile
           |
           +----java.awt.image.ByteComponentTile
```

---

public class **ByteComponentTile**
extends Tile

This class defines a tile with pixels consisting of one or more 8−bit channels stored in close proximity to each other in a byte array. The bit precision per channel element is that of the data type (that is, the bit precision for this tile is 8). There is only one pixel stride and one scanline stride for all channels.This type of tile can be used with a ComponentColorModel if there are multiple channels, or a IndexColorModel if there is only one channel.

The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Constructor Index

o **ByteComponentTile**(int, int, int, int, int, int, byte[])
    Constructs a base tile with 1 channel and a pixel stride of 1.
o **ByteComponentTile**(int, int, int, int, int, int[], int, int, byte[])
    Constructs a base tile given the channel array and tile dimensions.
o **ByteComponentTile**(int, int, int, int[], int, int, byte[])
    Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
    Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.
o **createCompatibleTile**(int, int)

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.

o **createSubtile**(int, int, int, int)
Creates a subtile given a region of the tile.
o **createSubtile**(int, int, int, int, int[])
Creates a subtile given a region of the tile.
o **getByteData**(int, int, int, int, byte[])
Returns a byte array of pixels from the specified rectangular region.
o **getByteData**(int, int, int, int, int, byte[])
Returns a byte array of pixels from the specified rectangular region.
o **getChannelOffset**(int)
Returns channel offset for the specified channel.
o **getChannelOffsets**()
Returns a copy of the channel offsets array.
o **getData**(int, int, int)
Returns the channel element for the specified channel at the specified location.
o **getData**(int, int, int, int, int, int[])
Returns an integer array of pixels from the specified rectangular region.
o **getData**(int, int, int, int, int[])
Returns an integer array of pixels from the specified rectangular region.
o **getData**(int, int, int[])
Returns the channel elements for all channels at the specified location.
o **getDataStorage**()
Returns a reference to the entire data array
o **getPixelStride**()
Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.
o **getScanlineStride**()
Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.
o **print**(int, int, int, int)
For debugging...
o **putByteData**(int, int, int, int, byte[])
Stores a byte array of pixels into the specified rectangular region.
o **putByteData**(int, int, int, int, int, byte[])
Stores a byte array of pixels into the specified rectangular region.
o **putData**(int, int, int, int)
Stores the channel element for the specified channel at the specified location.
o **putData**(int, int, int, int, int, int[])
Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int, int, int[])
Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int[])
Stores the channel elements for all channels at the specified location.
o **putData**(int, int, Tile)
Stores the Tile data at the specified location.
o **toString**()

Constructors

o **ByteComponentTile**

```
public ByteComponentTile(int width,
                         int height,
                         int numChannels,
                         int channelOffsets[],
                         int pixelStride,
                         int scanlineStride,
                         byte data[])
```

Constructs a base tile given the channel array and tile dimensions. The tile data will start at x=0, y=0 in the data array of each channel.

Parameters:
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile width and height.

o **ByteComponentTile**

```
public ByteComponentTile(int x,
                         int y,
                         int width,
                         int height,
                         int numChannels,
                         int channelOffsets[],
                         int pixelStride,
                         int scanlineStride,
                         byte data[])
```

Constructs a base tile given the channel array and tile dimensions.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.

pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

o **ByteComponentTile**

```
public ByteComponentTile(int x,
                         int y,
                         int width,
                         int height,
                         int channelOffset,
                         int scanlineStride,
                         byte data[])
```

Constructs a base tile with 1 channel and a pixel stride of 1.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
channelOffset – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getChannelOffsets**

```
public int[] getChannelOffsets()
```

Returns a copy of the channel offsets array.

o **getChannelOffset**

```
public int getChannelOffset(int channel)
```

Returns channel offset for the specified channel. The channel offset is the index into the channel data array in which the first channel element of the first scanline is stored.

o **getScanlineStride**

```
public int getScanlineStride()
```

Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **getPixelStride**

```
public int getPixelStride()
```

Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.

o **getDataStorage**

```
public byte[] getDataStorage()
```

Returns a reference to the entire data array

o **getData**

```
public int getData(int x,
                   int y,
                   int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int outData[])
```

Returns the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.

y – Y coordinate.
outData – If non–null, channel elements for all channels at the specified location are returned in this array.

Returns:

Data array with channel elements for all channels.

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int channel,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
int[] pixel = new int[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

Overrides:

getData in class Tile

o **getByteData**

```
public byte[] getByteData(int x,
                          int y,
                          int w,
                          int h,
                          int channel,
                          byte outData[])
```

Returns a byte array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
byte[] channelData = tile.getByteData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
byte chanElement = channelData[((y2-y)*w + (x2-x))];
```

o **getByteData**

```
public byte[] getByteData(int x,
                          int y,
                          int w,
                          int h,
                          byte outData[])
```

Returns a byte array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
byte[] channelData = tile.getByteData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
byte[] pixel = new byte[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

o **putData**

```
public void putData(int x,
                    int y,
                    int channel,
                    int inData)
```

Stores the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.
y – Y coordinate.
channel – Channel index.

inData – Channel element.

Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.
y – Y coordinate.
inData – Channel elements for all channels.

Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.
y – Y coordinate.
inTile – Tile of data to place at x,y location.

Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int channel,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are

assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putByteData**

```
public void putByteData(int x,
                        int y,
                        int w,
                        int h,
                        int channel,
                        byte inData[])
```

Stores a byte array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x)) + n]
```

o **putByteData**

```
public void putByteData(int x,
                        int y,
                        int w,
                        int h,
```

```
                          byte inData[])
```

Stores a byte array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width (in pixels) of the subtile
height − Height (in pixels) of the subtile
Throws: TileFormatException
if the specified bounding box is outside of the parent tile.
Overrides:
createSubtile in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height,
                          int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. A subset of the channels of the parent tile may be specified. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width (in pixels) of the subtile

height – Height (in pixels) of the subtile
channelList – Array of channel indices.

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Parameters that are guaranteed to remain the same are:

- number of channels
- pixel stride

Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
        may not be equal to:
gc.createCompatibleTile(w2, h2)
```

Overrides:
createCompatibleTile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays. If the tile is a subtile, this will call createCompatibleTile(width, height).

Overrides:
createCompatibleTile in class Tile

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

o **print**

```
public void print(int x,
                  int y,
                  int w,
                  int h)
```

For debugging... prints a region of a one−band ByteComponentTile

---

Class java.awt.image.ByteLookupTable

```
java.lang.Object
   |
   +----java.awt.image.LookupTable
           |
           +----java.awt.image.ByteLookupTable
```

---

public class **ByteLookupTable**
extends LookupTable

This class defines a lookup table object. The lookup table contains byte data for one or more tile channels or image components (for example, separate arrays for R, G, and B), and it contains an offset which will be subtracted from the input value before indexing the array. This allows an array smaller than the native data size to be constructed for a constrained input. If there is only one array in the lookup table, it will be applied to all channels.

See Also:
ShortLookupTable, LookupOp

Constructor Index

o **ByteLookupTable**(int, byte[])
Create a LookupTable object from an array of bytes representing a lookup table for each tile channel.
o **ByteLookupTable**(int, byte[][])
Create a LookupTable object from an array of byte arrays representing a lookup table for each tile channel.

Method Index

o **getDataStorage**()
Returns the lookup table data.
o **lookupPixel**(byte[], byte[])
Returns an array of components of a pixel, translated with the lookup table.
o **lookupPixel**(int[], int[])
Returns an array of components of a pixel, translated with the lookup table.

Constructors

o **ByteLookupTable**

```
public ByteLookupTable(int offset,
                       byte data[][])
```

Create a LookupTable object from an array of byte arrays representing a lookup table for each tile channel. The offset will be subtracted from the input value before indexing into the arrays. The number of components is the length of the data argument. The data array for each component stored as a reference.

o **ByteLookupTable**

```
public ByteLookupTable(int offset,
                       byte data[])
```

Create a LookupTable object from an array of bytes representing a lookup table for each tile channel. The offset will be subtracted from the input value before indexing into the array. The number of components is one. The data array will be stored internally.

Methods

o **getDataStorage**

```
public byte[][] getDataStorage()
```

Returns the lookup table data.

Returns:
LookupTable data array.

o **lookupPixel**

```
public int[] lookupPixel(int src[],
                         int dst[])
```

Returns an array of components of a pixel, translated with the lookup table. The source and destination can be equal. If dst is null, a new array will be allocated. The dst array is returned.

Returns:
an int array of components.
Overrides:
lookupPixel in class LookupTable

o **lookupPixel**

```
public byte[] lookupPixel(byte src[],
                          byte dst[])
```

Returns an array of components of a pixel, translated with the lookup table. The source and destination can be equal. If dst is null, a new array will be allocated. The dst array is returned.

Returns:
a byte array of components.

Class java.awt.image.BytePackedTile

```
java.lang.Object
   |
   +----java.awt.image.Tile
           |
           +----java.awt.image.BytePackedTile
```

public class **BytePackedTile**
extends Tile

This class is most useful for describing 1, 2 and 4 bit image tiles. This tile has one channel whose pixels are less than 8−bits and are packed together into individual bytes in a byte array. This type of tile can be used with an IndexColorModel. The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Constructor Index

o **BytePackedTile**(int, int, int, byte[])
    Constructs a tile given the width, height, and number of bits per pixel.
o **BytePackedTile**(int, int, int, int, int, int, int, int, byte[])
    Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
    Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.
o **createCompatibleTile**(int, int)
    Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.
o **createSubtile**(int, int, int, int)
    Creates a subtile given a region of the tile.
o **createSubtile**(int, int, int, int, int[])

Creates a subtile given a region of the tile.

o **getByteData**(int, int, int, int, byte[])
  Returns an array containing the specified channel elements from the data array.

o **getByteData**(int, int, int, int, int, byte[])
  Returns an array containing the specified channel elements from the data array.

o **getChannelBitOffset**()
  Returns channel bit offset for the specified channel.

o **getData**(int, int, int)
  Returns the channel element for the specified channel at the specified location.

o **getData**(int, int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.

o **getData**(int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.

o **getData**(int, int, int[])
  Returns the channel element at the specified location.

o **getDataStorage**()
  Returns a reference to the entire data array

o **getNumberOfBits**()
  Returns the total number of bits per channel element.

o **getPixelBitStride**()
  Returns pixel bit stride –– the number of bits between two channel elements on the same scanline.

o **getScanlineStride**()
  Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **putByteData**(int, int, int, int, byte[])
  Stores a byte array of pixels into the specified rectangular region.

o **putByteData**(int, int, int, int, int, byte[])
  Stores a byte array of pixels into the specified rectangular region.

o **putData**(int, int, int, int)
  Stores the channel element at the specified location.

o **putData**(int, int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.

o **putData**(int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.

o **putData**(int, int, int[])
  Stores the channel elements for all channels at the specified location.

o **putData**(int, int, Tile)
  Stores the Tile data at the specified location.

o **toString**()

Constructors

o **BytePackedTile**

```
public BytePackedTile(int width,
                      int height,
                      int numberOfBits,
                      byte data[])
```

Constructs a tile given the width, height, and number of bits per pixel. The data is assumed to be packed as tightly as numberOfBits allows, with no initial offset and with the smallest scanline stride to satisfy the constraint that adjacent rows differ by an integer number of byte array elements.

Parameters:

width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numberOfBits – Bits per pixel of the data
data – The data array.

Throws: TileFormatException

if the channel parameters are incompatible with the tile width and height. The verify() method is used to check consistency.

See Also:

verify

o **BytePackedTile**

```
public BytePackedTile(int x,
                      int y,
                      int width,
                      int height,
                      int numberOfBits,
                      int channelBitOffset,
                      int pixelBitStride,
                      int scanlineStride,
                      byte data[])
```

Constructs a base tile given the channel array and tile dimensions.

Parameters:

x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numberOfBits – Bit precision of the data
channelBitOffset – Bit index into the data array in which the first channel element of the first scanline is accessed.
pixelBitStride – Number of bits between two channel elements on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException

if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getNumberOfBits**

```
public int getNumberOfBits()
```

Returns the total number of bits per channel element.

o **getChannelBitOffset**

```
public int getChannelBitOffset()
```

Returns channel bit offset for the specified channel. The channel bit offset is the bit index into the channel data array in which the first channel element of the first scanline is stored.

o **getScanlineStride**

```
public int getScanlineStride()
```

Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **getPixelBitStride**

```
public int getPixelBitStride()
```

Returns pixel bit stride –– the number of bits between two channel elements on the same scanline.

o **getDataStorage**

```
public byte[] getDataStorage()
```

Returns a reference to the entire data array

o **getData**

```
public int getData(int x,
                   int y,
                   int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index is ignored..

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int outData[])
```

Returns the channel element at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.

y – Y coordinate.

outData – If non–null, channel element at the specified location is returned in this array.

Returns:

Data array with channel element

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int channel,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. The channel index will be ignored. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. The channel index will be ignored. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
int pixel;
// To find a pixel at location (x2, y2)
pixel = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getByteData**

```
public byte[] getByteData(int x,
                          int y,
                          int w,
                          int h,
                          int channel,
                          byte outData[])
```

Returns an array containing the specified channel elements from the data array. The channel index will be ignored. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
byte[] channelData = getByteData(x, y, channel, w, h, null);
// To find an element at location (x2, y2)
byte element = channelData[(y2-y)*w + (x2-x)];
```

o **getByteData**

```
public byte[] getByteData(int x,
                          int y,
                          int w,
                          int h,
                          byte outData[])
```

Returns an array containing the specified channel elements from the data array. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
byte[] channelData = tile.getByteData(x, y, w, h, null);
byte pixel;
// To find a pixel at location (x2, y2)
pixel = channelData[((y2-y)*w + (x2-x))];
```

o **putData**

```
public void putData(int x,
                    int y,
```

```
                    int channel,
                    int inData)
```

Stores the channel element at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index is ignored.
inData – Channel element.

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inData – Channel elements

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inTile – Tile of data to place at x,y location.

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int channel,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. The channel is ignored. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))]
```

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))]
```

Overrides:
putData in class Tile

o **putByteData**

```
public void putByteData(int x,
                        int y,
                        int w,
                        int h,
                        int channel,
                        byte inData[])
```

Stores a byte array of pixels into the specified rectangular region. The channel index will be ignored. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))]
```

o **putByteData**

```
public void putByteData(int x,
                        int y,
                        int w,
                        int h,
                        byte inData[])
```

Stores a byte array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))]
```

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width (in pixels) of the subtile
height − Height (in pixels) of the subtile
Throws: TileFormatException
if the specified bounding box is outside of the parent tile.
Overrides:
createSubtile in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height,
                          int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. A subset of the channels of the parent tile may be specified. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width (in pixels) of the subtile
height − Height (in pixels) of the subtile
channelList − Array of channel indices.

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Parameters that are guaranteed to remain the same are:

- number of bits
- pixel bit stride

Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
        may not be equal to:
gc.createCompatibleTile(w2, h2)
```

Overrides:
createCompatibleTile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.

Overrides:
createCompatibleTile in class Tile

o **toString**

```
public String toString()
```

Overrides:

toString in class Object

Class java.awt.image.ColorConvertOp

Class java.awt.image.ColorConvertOp

```
java.lang.Object
   |
   +----java.awt.image.ColorConvertOp
```

---

public class **ColorConvertOp**
extends Object
implements BufferedImageOp, TileOp

This class performs a pixel−by−pixel color conversion of the data in the source image. The resulting color values are scaled to the precision of the destination image data type. Color conversion can be specified via an array of ColorSpace objects or an array of ICC_Profile objects.

If the source is a BufferedImage with premultiplied alpha, the color components are divided by the alpha component before color conversion. If the destination is a BufferedImage with premultiplied alpha, the color components are multiplied by the alpha component after conversion. Tiles are treated as having no alpha channel, i.e. all channels are color channels.

Note Source and Destination may be the same object.

Constructor Index

o **ColorConvertOp**(ColorSpace[])
    Constructs a new ColorConvertOp from an array of ColorSpaces.
o **ColorConvertOp**(ICC_Profile[])
    Constructs a new ColorConvertOp from an array of ICC_Profiles.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
    Creates an empty destination image with the correct size and number of components, given this source.
o **createCompatibleDestTile**(Tile)
    Creates an empty destination tile with the correct size and number of channels, given this source.

o **filter**(BufferedImage, BufferedImage)
ColorConvert the BufferedImage.
o **filter**(Tile, Tile)
ColorConvert the channel data in the Tile.
o **getColorSpaces**()
Returns the array of ColorSpaces used to construct this ColorConvertOp.
o **getDestBounds**(BufferedImage)
Returns the bounding box of the destination, given this source.
o **getDestBounds**(Tile)
Returns the bounding box of the destination, given this source.
o **getDestPoint**(Point2D, Point2D)
Returns the location of the destination point given a point in the source image.
o **getICC_Profiles**()
Returns the array of ICC_Profiles used to construct this ColorConvertOp.

Constructors

o **ColorConvertOp**

```
public ColorConvertOp(ColorSpace cspaces[])
```

Constructs a new ColorConvertOp from an array of ColorSpaces. The semantics is that source pixels are converted in sequence between successive ColorSpaces in the array, i.e. from cspaces[0] to cspaces[1], then from cspaces[1] to cspaces[2], etc.

For BufferedImages, if the ColorSpace of the source BufferedImage does not match the first ColorSpace in the array, the first conversion is to the first ColorSpace in the array. If the last ColorSpace in the array does not match the ColorSpace of the destination BufferedImage, the last conversion is to the destination's ColorSpace. For BufferedImages, the semantics are well−defined for ColorConvertOps constructed from an array of any length.

For Tiles, the number of channels in the source Tile must match the number of components of the first ColorSpace in the array, and the number of channels in the destination Tile must match the number of components in the last ColorSpace in the array. For arrays of length zero or one, the ColorConvertOp is a NoOp for Tiles.

o **ColorConvertOp**

```
public ColorConvertOp(ICC_Profile profiles[])
```

Constructs a new ColorConvertOp from an array of ICC_Profiles. The sequence of profiles may include profiles that represent color spaces, profiles that represent effects, etc. If the whole sequence does not represent a well−defined color conversion, an exception is thrown.

For BufferedImages, if the ColorSpace of the source BufferedImage does not match the requirements of the first profile in the array, the first conversion is to an appropriate ColorSpace. If the requirements of the last profile in the array are not met by the ColorSpace of the destination BufferedImage, the last conversion is to the destination's ColorSpace.

For Tiles, the number of channels in the source Tile must match the requirements of the first profile in the array, and the number of channels in the destination Tile must match the requirements of the last profile in the array.

Throws: IllegalArgumentException
: when the profile sequence does not specify a well−defined color conversion

Methods

o **getColorSpaces**

```
public ColorSpace[] getColorSpaces()
```

Returns the array of ColorSpaces used to construct this ColorConvertOp. Returns null if the ColorConvertOp was constructed from an array of ICC_Profiles.

o **getICC_Profiles**

```
public ICC_Profile[] getICC_Profiles()
```

Returns the array of ICC_Profiles used to construct this ColorConvertOp. Returns null if the ColorConvertOp was constructed from an array of ColorSpaces.

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dest)
```

ColorConvert the BufferedImage. If the destination image is null, a BufferedImage will be created with an appropriate ColorModel.

o **filter**

```
public Tile filter(Tile src,
                   Tile dest)
```

ColorConvert the channel data in the Tile. If the destination Tile is null, a new Tile will be created. The number of channels in the source and destination Tiles must meet the requirements explained above. Otherwise, an exception is thrown.

Throws: IllegalArgumentException
: if the number of source or destination channels is incorrect

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the destination, given this source. Note that this will be the same as the the bounding box of the source.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the destination, given this source. Note that this will be the same as the the bounding box of the source.

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of components, given this source.

Parameters:
src – Source image for the filter operation
destCM – ColorModel of the destination. If null, an appropriate ColorModel will be used.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels, given this source.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                            Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non–null, it will be used to hold the return value.

Class java.awt.image.ColorModel

```
java.lang.Object
   |
   +----java.awt.image.ColorModel
```

---

public abstract class **ColorModel**
extends Object
implements Transparency
super class of:

ComponentColorModel, IndexColorModel, PackedColorModel

An abstract class that encapsulates the methods for translating from pixel values to color components (e.g. red, green, blue) for an image. Pixels may be represented as 32−bit ints or as arrays of 32−bit ints.

See Also:

IndexColorModel, ComponentColorModel, PackedColorModel, DirectColorModel

---

Variable Index

o **pixel_bits**

Constructor Index

o **ColorModel**(int)

Constructs a ColorModel which describes a pixel of the specified number of bits.

o **ColorModel**(int, ColorSpace, boolean, boolean, int)

Constructs a ColorModel which describes a pixel of the specified number of bits in the color space of the specified ColorSpace.

o **ColorModel**(int[], ColorSpace, boolean, boolean, int)

Constructs a ColorModel which describes a pixel in the color space of the specified ColorSpace.

Method Index

o **coerceData**(Tile, boolean)
  Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel.
o **createCompatibleTile**(int, int)
  Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.
o **equals**(Object)
o **finalize**()
o **getAlpha**(int)
  Returns the alpha component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getAlpha**(int[])
  Returns the alpha color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getBlue**(int)
  Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getBlue**(int[])
  Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getColorSpace**()
  Returns the ColorSpace associated with this ColorModel
o **getComponents**(float[], int[])
  Returns an array of all of the components given a normalized component array.
o **getComponents**(int, int[])
  Returns an array of all of the components given a pixel in the current color model.
o **getComponents**(int[], int[])
  Returns an array of all of the components given a pixel array in the current color model.
o **getComponentSize**()
  Returns an array of the number of bits per component.
o **getComponentSize**(int)
  Returns the number of bits for the given component.
o **getGreen**(int)
  Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getGreen**(int[])
  Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getNormalizedComponents**(int[], float[])
  Returns an array of normalized components given an array of all components.
o **getNumColorComponents**()
  Returns the number of color components in this color model.
o **getNumComponents**()
  Returns the number of components (including alpha) in this color model.
o **getPixel**(int, int[])
  Returns a pixel array given a pixel in the default RGB color model.
o **getPixelSize**()

Returns the number of bits per pixel described by this ColorModel.

o **getRed**(int)
: Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getRed**(int[])
: Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getRGB**(int)
: Returns the color of the pixel in the default RGB color model.

o **getRGB**(int[])
: Returns a pixel in the default RGB color model given a pixel array in the current color model.

o **getRGBdefault**()
: Return a DirectColorModel that describes the default format for integer RGB values used in many of the methods in the AWT image interfaces for the convenience of the programmer.

o **getTransparency**()
: Returns the transparency.

o **hasAlpha**()
: Returns whether or not alpha is supported in this ColorModel.

o **isAlphaPremultiplied**()
: Returns whether or not the alpha has been premultiplied in the Tile associated with this ColorModel.

o **toString**()
: Prints the contents of this object

Variables

o **pixel_bits**

```
protected int pixel_bits
```

Constructors

o **ColorModel**

```
protected ColorModel(int bits)
```

Constructs a ColorModel which describes a pixel of the specified number of bits. The color space is the default RGB ColorSpace, which is sRGB.

o **ColorModel**

```
protected ColorModel(int bits,
                     ColorSpace cspace,
                     boolean hasAlpha,
                     boolean isAlphaPremultiplied,
                     int transparency)
```

Constructs a ColorModel which describes a pixel of the specified number of bits in the color space of the specified ColorSpace. The boolean, isAlphaPremultiplied, specifies how to interpret color data that are passed to this ColorModel. If the boolean is true, the color data have been multiplied with alpha. The transparency specifies what alpha values can be represented by this color model. Any ColorModel object that uses this constructor should override the getComponentArray() method to return the actual bits per component array. This constructor is here to support legacy code.

See Also:
Transparency

o **ColorModel**

```
protected ColorModel(int bits[],
                     ColorSpace cspace,
                     boolean hasAlpha,
                     boolean isAlphaPremultiplied,
                     int transparency)
```

Constructs a ColorModel which describes a pixel in the color space of the specified ColorSpace. The bits array specifies the number of significant bits per component in the color model (color components and alpha). The boolean, isAlphaPremultiplied, specifies how to interpret color data that are passed to this ColorModel. If the boolean is true, the color data have been multiplied with alpha. The transparency specifies what alpha values can be represented by this color model.

See Also:
Transparency

Methods

o **getRGBdefault**

```
public static ColorModel getRGBdefault()
```

Return a DirectColorModel that describes the default format for integer RGB values used in many of the methods in the AWT image interfaces for the convenience of the programmer. The color space is the default ColorSpace, sRGB. The format for the RGB values is an integer with 8 bits each of alpha, red, green, and blue color components ordered correspondingly from the most significant byte to the least significant byte, as in: 0xAARRGGBB. Color components are not premultiplied by the alpha component. This format does not necessarily represent the native or the most efficient ColorModel for a particular device or for all images. It is merely used as a common color model format.

o **hasAlpha**

```
public boolean hasAlpha()
```

Returns whether or not alpha is supported in this ColorModel.

o **isAlphaPremultiplied**

```
public boolean isAlphaPremultiplied()
```

Returns whether or not the alpha has been premultiplied in the Tile associated with this ColorModel.

o **getPixelSize**

```
public int getPixelSize()
```

Returns the number of bits per pixel described by this ColorModel.

o **getComponentSize**

```
public int getComponentSize(int componentIdx)
```

Returns the number of bits for the given component.

o **getComponentSize**

```
public int[] getComponentSize()
```

Returns an array of the number of bits per component.

o **getTransparency**

```
public int getTransparency()
```

Returns the transparency. Returns either OPAQUE, BITMASK, or TRANSLUCENT

See Also:
OPAQUE, BITMASK, TRANSLUCENT

o **getNumComponents**

```
public int getNumComponents()
```

Returns the number of components (including alpha) in this color model. This is equal to the number of color components, optionally plus one, if there is an alpha component.

o **getNumColorComponents**

```
public int getNumColorComponents()
```

Returns the number of color components in this color model. This is the number of components returned by ColorSpace.getNumComponents().

See Also:
getNumComponents

o **getRed**

```
public abstract int getRed(int pixel)
```

Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

o **getGreen**

```
public abstract int getGreen(int pixel)
```

Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

o **getBlue**

```
public abstract int getBlue(int pixel)
```

Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

o **getAlpha**

```
public abstract int getAlpha(int pixel)
```

Returns the alpha component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel.

o **getRGB**

```
public int getRGB(int pixel)
```

Returns the color of the pixel in the default RGB color model. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

See Also:
getRGBdefault

o **getRed**

```
public int getRed(int pixel[])
```

Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

o **getGreen**

```
public int getGreen(int pixel[])
```

Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

o **getBlue**

```
public int getBlue(int pixel[])
```

Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

o **getAlpha**

```
public int getAlpha(int pixel[])
```

Returns the alpha color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel.

o **getRGB**

```
public int getRGB(int pixel[])
```

Returns a pixel in the default RGB color model given a pixel array in the current color model. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

See Also:
getRGBdefault

o **getPixel**

```
public int[] getPixel(int rgb,
                      int pixel[])
```

Returns a pixel array given a pixel in the default RGB color model. This pixel array can then be passed to the putData method of a Tile object. If the pixel variable is null, a new array will be allocated. The pixel array will be returned.

o **getComponents**

```
public int[] getComponents(int pixel,
                           int components[])
```

Returns an array of all of the components given a pixel in the current color model. If the color model is not an IndexColorModel or DirectColorModel, an IllegalArgumentException will be thrown. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

o **getComponents**

```
public int[] getComponents(int pixel[],
                           int components[])
```

Returns an array of all of the components given a pixel array in the current color model. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

o **getComponents**

```
public int[] getComponents(float normComponents[],
                           int components[])
```

Returns an array of all of the components given a normalized component array.

o **getNormalizedComponents**

```
public float[] getNormalizedComponents(int components[],
                                       float normComponents[])
```

Returns an array of normalized components given an array of all components.

o **equals**

```
public boolean equals(Object obj)
```

Overrides:
equals in class Object

o **finalize**

```
public void finalize()
```

Overrides:
finalize in class Object

o **getColorSpace**

```
public ColorSpace getColorSpace()
```

Returns the ColorSpace associated with this ColorModel

o **coerceData**

```
public ColorModel coerceData(Tile tile,
                             boolean isAlphaPremultiplied)
```

Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel. It may multiply or divide the color Tile data by alpha, or do nothing if the data is in the correct state. If the data needs to be coerced, this method will also return an instance of this ColorModel with the isAlphaPremultiplied flag set appropriately.

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.

See Also:
Tile

o **toString**

```
public String toString()
```

Prints the contents of this object

Overrides:
toString in class Object

Class java.awt.image.ComponentColorModel

```
java.lang.Object
   |
   +----java.awt.image.ColorModel
           |
           +----java.awt.image.ComponentColorModel
```

public class **ComponentColorModel**
extends ColorModel

A ColorModel class 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.

See Also:
    ColorModel

Constructor Index

o **ComponentColorModel**(ColorSpace, int[], boolean, boolean, int)
    Constructs a ComponentColorModel from the specified parameters.

Method Index

o **coerceData**(Tile, boolean)
    Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel.
o **createCompatibleTile**(int, int)
    Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.
o **equals**(Object)
    Compares this color model with another for equality.
o **getAlpha**(int)
    Returns the alpha color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getAlpha**(int[])

Returns the alpha component for the specified pixel, scaled from 0 to 255.

o **getBlue**(int)
: Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getBlue**(int[])
: Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getComponents**(int, int[])
: Returns an array of all of the components given a pixel in the current color model.

o **getComponents**(int[], int[])
: Returns an array of all of the components given a pixel array in the current color model.

o **getGreen**(int)
: Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getGreen**(int[])
: Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getPixel**(int, int[])
: Returns a pixel array given a pixel in the default RGB color model.

o **getRed**(int)
: Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getRed**(int[])
: Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.

o **getRGB**(int)
: Returns the color of the pixel in the default RGB color model.

o **getRGB**(int[])
: Returns a pixel in the default RGB color model given a pixel array in the current color model.

Constructors

o **ComponentColorModel**

```
 public ComponentColorModel(ColorSpace colorSpace,
                            int bits[],
                            boolean hasAlpha,
                            boolean isAlphaPremultiplied,
                            int transparency)
```

Constructs a ComponentColorModel from the specified parameters. The bits array specifies the number of significant bits per component in the color model (color components and alpha). An IllegalArgumentException will be thrown if the length of the array does not match the number of components. If hasAlpha is true, then the boolean, isAlphaPremultiplied, specifies how to interpret color data that are passed to this ColorModel. If it is true, the color data will have been multiplied with alpha.

Parameters:
: colorSpace − ColorSpace associated with this color model

bits – Number of significant bits per component
hasAlpha – If true, this color model supports alpha.
isAlphaPremultiplied – If true, alpha is premultiplied.
transparency – Specifies what alpha values can be represented by this color model.

See Also:
ColorSpace, Transparency

Methods

o **getRed**

```
public int getRed(int pixel)
```

Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel.

Overrides:
getRed in class ColorModel

o **getGreen**

```
public int getGreen(int pixel)
```

Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel.

Overrides:
getGreen in class ColorModel

o **getBlue**

```
public int getBlue(int pixel)
```

Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel.

Overrides:
getBlue in class ColorModel

o **getAlpha**

```
public int getAlpha(int pixel)
```

Returns the alpha color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel.

Overrides:
getAlpha in class ColorModel

o **getRGB**

```
public int getRGB(int pixel)
```

Returns the color of the pixel in the default RGB color model. An IllegalArgumentException will be thrown if there is more than one component in this ColorModel. This method will throw an IllegalArgumentException since this ColorModel is unable to extract alpha and color components from a single integer pixel.

Overrides:
getRGB in class ColorModel

o **getRed**

```
public int getRed(int pixel[])
```

Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getRed in class ColorModel

o **getGreen**

```
public int getGreen(int pixel[])
```

Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getGreen in class ColorModel

o **getBlue**

```
public int getBlue(int pixel[])
```

Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default

RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getBlue in class ColorModel

o **getAlpha**

```
public int getAlpha(int pixel[])
```

Returns the alpha component for the specified pixel, scaled from 0 to 255. The pixel is described by a discrete array of channel elements, one per channel.

Overrides:
getAlpha in class ColorModel

o **getRGB**

```
public int getRGB(int pixels[])
```

Returns a pixel in the default RGB color model given a pixel array in the current color model. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getRGB in class ColorModel
See Also:
getRGBdefault

o **getPixel**

```
public int[] getPixel(int rgb,
                      int pixel[])
```

Returns a pixel array given a pixel in the default RGB color model. This pixel array can then be passed to the putData method of a Tile object. If the pixel variable is null, a new array will be allocated. The pixel array will be returned.

Overrides:
getPixel in class ColorModel

o **getComponents**

```
public int[] getComponents(int pixel,
                           int components[])
```

Returns an array of all of the components given a pixel in the current color model. If the color model is not an IndexColorModel or DirectColorModel, an IllegalArgumentException will be thrown. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

Overrides:
getComponents in class ColorModel

o **getComponents**

```
public int[] getComponents(int pixel[],
                           int components[])
```

Returns an array of all of the components given a pixel array in the current color model. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

Overrides:
getComponents in class ColorModel

o **coerceData**

```
public ColorModel coerceData(Tile tile,
                             boolean isAlphaPremultiplied)
```

Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel. It may multiply or divide the color Tile data by alpha, or do nothing if the data is in the correct state. If the data needs to be coerced, this method will also return an instance of this ColorModel with the isAlphaPremultiplied flag set appropriately.

Overrides:
coerceData in class ColorModel

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.

Overrides:
createCompatibleTile in class ColorModel
See Also:
Tile

o **equals**

```
public boolean equals(Object obj)
```

Compares this color model with another for equality.

Overrides:
equals in class ColorModel

Class java.awt.image.ConvolveOp

All Packages Class Hierarchy This Package Previous Next Index

---

Class java.awt.image.ConvolveOp

```
java.lang.Object
   |
   +----java.awt.image.ConvolveOp
```

---

public class **ConvolveOp**
extends Object
implements BufferedImageOp, TileOp

This class implements a convolution from the source to the destination. Convolution using a convolution kernel is a spatial operation that computes the output pixel from an input pixel by multiplying the kernel with the surround of the input pixel. This allows the output pixel to be affected by the immediate neighborhood in a way that can be mathematically specified with a kernel.

This class operates with BufferedImage data that are premultiplied with alpha. If the Source BufferedImage has an alpha component, and the color components are not premultiplied with the alpha component, then the data are premultiplied before being convolved. If the Destination has alpha components which are not premultiplied, then alpha is divided out before storing into the Destination. If the Destination has no alpha component, then the resulting alpha is discarded after first dividing it out of the color components.

Tiles are treated as having no alpha channel.

Note that the Source and the Destination may not be the same object.

See Also:
Kernel

Variable Index

o **EDGE_EXTEND**
The convolution operator temporarily extends the width and height of the source image by replicating the pixels at the edge of the image.
o **EDGE_NO_OP**
Values of pixels at the edge of the source image are copied to the corresponding pixels in the destination without modification.

o **EDGE_ZERO_FILL**
Pixels at the edge of the destination image are set to zero.

Constructor Index

o **ConvolveOp**(Kernel, int)
Constructs a ConvolveOp given a Kernel and an edge hint.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
Creates an empty destination image with the correct size and number of channels.
o **createCompatibleDestTile**(Tile)
Creates an empty destination tile with the correct size and number of channels.
o **filter**(BufferedImage, BufferedImage)
Perform a convolution on BufferedImages.
o **filter**(Tile, Tile)
Perform a convolution on Tiles.
o **getDestBounds**(BufferedImage)
Returns the bounding box of the filtered destination image.
o **getDestBounds**(Tile)
Returns the bounding box of the filtered destination tile.
o **getDestPoint**(Point2D, Point2D)
Returns the location of the destination point given a point in the source image.
o **getEdgeHint**()
Return the edge hint.
o **getKernel**()
Return the Kernel.

Variables

o **EDGE_NO_OP**

```
public static final int EDGE_NO_OP
```

Values of pixels at the edge of the source image are copied to the corresponding pixels in the destination without modification. Thus the edge of the source is not filtered.

o **EDGE_ZERO_FILL**

```
public static final int EDGE_ZERO_FILL
```

Pixels at the edge of the destination image are set to zero.

o **EDGE_EXTEND**

```
public static final int EDGE_EXTEND
```

The convolution operator temporarily extends the width and height of the source image by replicating the pixels at the edge of the image. This strategy enables the operator to filter the pixels that were originally at the edge of the image.

Constructors

o **ConvolveOp**

```
public ConvolveOp(Kernel kernel,
                  int edgeHint)
```

Constructs a ConvolveOp given a Kernel and an edge hint.

See Also:
Kernel, EDGE_NO_OP, EDGE_ZERO_FILL, EDGE_EXTEND

Methods

o **getEdgeHint**

```
public int getEdgeHint()
```

Return the edge hint.

See Also:
EDGE_NO_OP, EDGE_ZERO_FILL, EDGE_EXTEND

o **getKernel**

```
public Kernel getKernel()
```

Return the Kernel.

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst)
```

Perform a convolution on BufferedImages. Each component of the source image will be convolved. If the color model in the source image is not the same as that in the destination image, the pixels will be converted in the destination. If the destination image is null, a BufferedImage will be created with the source ColorModel. The IllegalArgumentException may be thrown if the source is the same as the destination.

o **filter**

```
public Tile filter(Tile src,
                   Tile dst)
```

Perform a convolution on Tiles. Each channel of the source tile will be convolved. The source and destination must have the same number of channels. If the destination Tile is null, a new Tile will be created. The IllegalArgumentException may be thrown if the source is the same as the destination.

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of channels.

Parameters:

src − Source image for the filter operation
destCM − ColorModel of the destination. If null, the ColorModel of the source will be used.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels.

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the filtered destination image. Since this is not a geometric operation, the bounding box does not change.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the filtered destination tile. Since this is not a geometric operation, the bounding box does not change.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                            Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non−null, it will be used to hold the return value. Since this is not a geometric operation, the srcPt will equal the dstPt.

Class java.awt.image.DirectColorModel

```
java.lang.Object
   |
   +----java.awt.image.ColorModel
           |
           +----java.awt.image.PackedColorModel
                   |
                   +----java.awt.image.DirectColorModel
```

public class **DirectColorModel**
extends PackedColorModel

A ColorModel class that represents pixel values that have RGB color components embedded directly in the bits of the pixel itself. This color model is similar to an X11 TrueColor visual. The default RGB ColorModel specified by the ColorModel.getRGBdefault method is a DirectColorModel with the following parameters:

```
 Number of bits:        32
 Red mask:              0x00ff0000
 Green mask:            0x0000ff00
 Blue mask:             0x000000ff
 Alpha mask:            0xff000000
 Color space:           sRGB
 isAlphaPremultiplied:  False
```

Many of the methods in this class are final. This is because the underlying native graphics code makes assumptions about the layout and operation of this class and those assumptions are reflected in the implementations of the methods here that are marked final. You can subclass this class for other reasons, but you cannot override or modify the behaviour of those methods.

See Also:
ColorModel, ColorSpace

Constructor Index

o **DirectColorModel**(ColorSpace, int, int, int, int, int, boolean)
Constructs a DirectColorModel from the specified parameters.

o **DirectColorModel**(int, int, int, int)
  Constructs a DirectColorModel from the given masks specifying which bits in the pixel contain the red, green and blue color components.
o **DirectColorModel**(int, int, int, int, int)
  Constructs a DirectColorModel from the given masks specifying which bits in the pixel contain the alpha, red, green and blue color components.

Method Index

o **coerceData**(Tile, boolean)
  Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel.
o **createCompatibleTile**(int, int)
  Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.
o **getAlpha**(int)
  Return the alpha transparency value for the specified pixel in the range 0–255.
o **getAlpha**(int[])
  Returns the alpha color component for the specified pixel, scaled from 0 to 255.
o **getAlphaMask**()
  Returns the mask indicating which bits in a pixel contain the alpha transparency component.
o **getBlue**(int)
  Returns the blue color component for the specified pixel in the range 0–255 in the default RGB ColorSpace, sRGB.
o **getBlue**(int[])
  Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getBlueMask**()
  Returns the mask indicating which bits in a pixel contain the blue color component.
o **getComponents**(int, int[])
  Returns an array of all of the components given a pixel in the current color model.
o **getComponents**(int[], int[])
  Returns an array of all of the components given a pixel array in the current color model.
o **getGreen**(int)
  Returns the green color component for the specified pixel in the range 0–255 in the default RGB ColorSpace, sRGB.
o **getGreen**(int[])
  Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB.
o **getGreenMask**()
  Returns the mask indicating which bits in a pixel contain the green color component.
o **getPixel**(int, int[])
  Returns a pixel array given a pixel in the default RGB color model.
o **getRed**(int)
  Returns the red color component for the specified pixel in the range 0–255 in the default RGB ColorSpace, sRGB.
o **getRed**(int[])
  Returns the red color component for the specified pixel, scaled from 0 to 255 in the default

RGB ColorSpace, sRGB.

o **getRedMask**()
: Returns the mask indicating which bits in a pixel contain the red color component.

o **getRGB**(int)
: Returns the color of the pixel in the default RGB color model.

o **getRGB**(int[])
: Returns a pixel in the default RGB color model given a pixel array in the current color model.

o **toString**()
: Prints the contents of this object

Constructors

o **DirectColorModel**

```
public DirectColorModel(int bits,
                        int rmask,
                        int gmask,
                        int bmask)
```

Constructs a DirectColorModel from the given masks specifying which bits in the pixel contain the red, green and blue color components. Pixels described by this color model will all have alpha components of 255 (fully opaque). All of the bits in each mask must be contiguous and fit in the specified number of least significant bits of the integer.

o **DirectColorModel**

```
public DirectColorModel(int bits,
                        int rmask,
                        int gmask,
                        int bmask,
                        int amask)
```

Constructs a DirectColorModel from the given masks specifying which bits in the pixel contain the alpha, red, green and blue color components. All of the bits in each mask must be contiguous and fit in the specified number of least significant bits of the integer. The ColorSpace will be the default sRGB space, and alpha, if present, will not be premultiplied.

o **DirectColorModel**

```
public DirectColorModel(ColorSpace space,
                        int bits,
                        int rmask,
                        int gmask,
                        int bmask,
                        int amask,
                        boolean isAlphaPremultiplied)
```

Constructs a DirectColorModel from the specified parameters. The masks specify which bits in the pixel contain the alpha, red, green and blue color components. All of the bits in each mask must be contiguous and fit in the specified number of least significant bits of the integer. The ColorSpace of the color model is specified by space and must be of ColorSpace.TYPE_RGB. If there is alpha, the boolean, isAlphaPremultiplied, specifies how to interpret color data that are passed to this ColorModel. If it is true, the color data will have been multiplied with alpha.

Methods

o **getRedMask**

```
public final int getRedMask()
```

Returns the mask indicating which bits in a pixel contain the red color component.

o **getGreenMask**

```
public final int getGreenMask()
```

Returns the mask indicating which bits in a pixel contain the green color component.

o **getBlueMask**

```
public final int getBlueMask()
```

Returns the mask indicating which bits in a pixel contain the blue color component.

o **getAlphaMask**

```
public final int getAlphaMask()
```

Returns the mask indicating which bits in a pixel contain the alpha transparency component.

o **getRed**

```
public final int getRed(int pixel)
```

Returns the red color component for the specified pixel in the range 0−255 in the default RGB ColorSpace, sRGB. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getRed in class ColorModel

o **getGreen**

```
public final int getGreen(int pixel)
```

Returns the green color component for the specified pixel in the range 0−255 in the default RGB ColorSpace, sRGB. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is

0, the color will be 0.

Overrides:
getGreen in class ColorModel

o **getBlue**

```
public final int getBlue(int pixel)
```

Returns the blue color component for the specified pixel in the range 0−255 in the default RGB ColorSpace, sRGB. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getBlue in class ColorModel

o **getAlpha**

```
public final int getAlpha(int pixel)
```

Return the alpha transparency value for the specified pixel in the range 0−255.

Overrides:
getAlpha in class ColorModel

o **getRGB**

```
public final int getRGB(int pixel)
```

Returns the color of the pixel in the default RGB color model. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned).

Overrides:
getRGB in class ColorModel
See Also:
getRGBdefault

o **getRed**

```
public int getRed(int pixel[])
```

Returns the red color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getRed in class ColorModel

o **getGreen**

```
public int getGreen(int pixel[])
```

Returns the green color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getGreen in class ColorModel

o **getBlue**

```
public int getBlue(int pixel[])
```

Returns the blue color component for the specified pixel, scaled from 0 to 255 in the default RGB ColorSpace, sRGB. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getBlue in class ColorModel

o **getAlpha**

```
public int getAlpha(int pixel[])
```

Returns the alpha color component for the specified pixel, scaled from 0 to 255. The pixel is described by a discrete array of channel elements, one per channel.

Overrides:
getAlpha in class ColorModel

o **getRGB**

```
public int getRGB(int pixels[])
```

Returns a pixel in the default RGB color model given a pixel array in the current color model. The pixel is described by a discrete array of channel elements, one per channel. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getRGB in class ColorModel
See Also:
getRGBdefault

o **getPixel**

```
public int[] getPixel(int rgb,
                      int pixel[])
```

Returns a pixel array given a pixel in the default RGB color model. This pixel array can then be passed to the putData method of a Tile object. If the pixel variable is null, a new array will be allocated. The pixel array will be returned.

Overrides:
getPixel in class ColorModel

o **getComponents**

```
public final int[] getComponents(int pixel,
                                 int components[])
```

Returns an array of all of the components given a pixel in the current color model. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

Overrides:
getComponents in class ColorModel

o **getComponents**

```
public final int[] getComponents(int pixel[],
                                 int components[])
```

Returns an array of all of the components given a pixel array in the current color model. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

Overrides:
getComponents in class ColorModel

o **createCompatibleTile**

```
public final Tile createCompatibleTile(int w,
                                       int h)
```

Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.

Overrides:
: createCompatibleTile in class ColorModel

See Also:
: Tile

o **coerceData**

```
public final ColorModel coerceData(Tile tile,
                                   boolean isAlphaPremultiplied)
```

Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel. It may multiply or divide the color Tile data by alpha, or do nothing if the data is in the correct state. If the data needs to be coerced, this method will also return an instance of this ColorModel with the isAlphaPremultiplied flag set appropriately.

Overrides:
: coerceData in class ColorModel

o **toString**

```
public String toString()
```

Prints the contents of this object

Overrides:
: toString in class ColorModel

Class java.awt.image.IndexColorModel

All Packages Class Hierarchy This Package Previous Next Index

---

Class java.awt.image.IndexColorModel

```
java.lang.Object
   |
   +----java.awt.image.ColorModel
           |
           +----java.awt.image.IndexColorModel
```

---

public class **IndexColorModel**
extends ColorModel

A ColorModel class that represents pixel values which are indices into a fixed colormap in the ColorModel's color space. An optional transparent pixel value can be supplied which indicates a completely transparent pixel, regardless of any alpha value recorded for that pixel value. This color model is similar to an X11 PseudoColor visual. Note that alpha values in IndexColorModels are never premultiplied.

Many of the methods in this class are final. The reason for this is that the underlying native graphics code makes assumptions about the layout and operation of this class and those assumptions are reflected in the implementations of the methods here that are marked final. You can subclass this class for other reaons, but you cannot override or modify the behaviour of those methods.

See Also:
    ColorModel

Constructor Index

o **IndexColorModel**(int, int, byte[], byte[], byte[])
    Constructs an IndexColorModel from the given arrays of red, green, and blue components.
o **IndexColorModel**(int, int, byte[], byte[], byte[], byte[])
    Constructs an IndexColorModel from the given arrays of red, green, blue and alpha components.
o **IndexColorModel**(int, int, byte[], byte[], byte[], int)
    Constructs an IndexColorModel from the given arrays of red, green, and blue components.
o **IndexColorModel**(int, int, byte[], int, boolean)
    Constructs an IndexColorModel from a single array of packed red, green, blue and optional alpha components.
o **IndexColorModel**(int, int, byte[], int, boolean, int)

Constructs an IndexColorModel from a single array of packed red, green, blue and optional alpha components.

o **IndexColorModel**(int, int, int[], int, boolean, int)
  Constructs an IndexColorModel from an array of integers where each integer is comprised of red, green, blue, and optional alpha components in the default RGB color model packing.

Method Index

o **coerceData**(Tile, boolean)
  Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel.

o **convertToIntDiscrete**(Tile, boolean)
  Returns a new BufferedImage of TYPE_INT_ARGB or TYPE_INT_RGB that matches the lookup tables.

o **createCompatibleTile**(int, int)
  Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.

o **getAlpha**(int)
  Return the alpha transparency value for the specified pixel in the range 0–255.

o **getAlphas**(byte[])
  Copies the array of alpha transparency values into the given array.

o **getBlue**(int)
  Returns the blue color component for the specified pixel in the range 0–255 in the default RGB ColorSpace, sRGB.

o **getBlues**(byte[])
  Copies the array of blue color components into the given array.

o **getComponents**(int, int[])
  Returns an array of all of the components given a pixel in the current color model.

o **getComponents**(int[], int[])
  Returns an array of all of the components given a pixel array in the current color model.

o **getComponentSize**()
  Returns an array of the number of bits per component.

o **getDataArray**(int, byte[])
  Copies the nth data array (color or alpha component) into the given array.

o **getGreen**(int)
  Returns the green color component for the specified pixel in the range 0–255 in the default RGB ColorSpace, sRGB.

o **getGreens**(byte[])
  Copies the array of green color components into the given array.

o **getMapSize**()
  Returns the size of the color component arrays in this IndexColorModel.

o **getPixel**(int, int[])
  Returns a pixel array given a pixel in the default RGB color model.

o **getRed**(int)
  Returns the red color component for the specified pixel in the range 0–255 in the default RGB ColorSpace, sRGB.

o **getReds**(byte[])
  Copies the array of red color components into the given array.

o **getRGB**(int)
Returns the color of the pixel in the default RGB color model.
o **getRGBs**(int[])
Converts the arrays color and alpha components to the default RGB ColorModel (if necessary) and copies the resulting array of 32−bit ARGB values into the given array.
o **getTransparency**()
Returns the transparency.
o **getTransparentPixel**()
Returns the index of the transparent pixel in this IndexColorModel or −1 if there is no transparent pixel.

Constructors

o **IndexColorModel**

```
public IndexColorModel(int bits,
                       int size,
                       byte r[],
                       byte g[],
                       byte b[])
```

Constructs an IndexColorModel from the given arrays of red, green, and blue components. Pixels described by this color model will all have alpha components of 255 (fully opaque). All of the arrays specifying the color components must have at least the specified number of entries. Uses the default sRGB ColorSpace.

Parameters:
bits − The number of bits each pixel occupies.
size − The size of the color component arrays.
r − The array of red color components.
g − The array of green color components.
b − The array of blue color components.

o **IndexColorModel**

```
public IndexColorModel(int bits,
                       int size,
                       byte r[],
                       byte g[],
                       byte b[],
                       int trans)
```

Constructs an IndexColorModel from the given arrays of red, green, and blue components. Pixels described by this color model will all have alpha components of 255 (fully opaque), except for the indicated transparent pixel. All of the arrays specifying the color components must have at least the specified number of entries. Uses the default sRGB ColorSpace.

Parameters:
bits − The number of bits each pixel occupies.
size − The size of the color component arrays.

r – The array of red color components.
g – The array of green color components.
b – The array of blue color components.
trans – The index of the transparent pixel.

o **IndexColorModel**

```
public IndexColorModel(int bits,
                       int size,
                       byte r[],
                       byte g[],
                       byte b[],
                       byte a[])
```

Constructs an IndexColorModel from the given arrays of red, green, blue and alpha components. All of the arrays specifying the color components must have at least the specified number of entries. Uses the default sRGB ColorSpace.

Parameters:

bits – The number of bits each pixel occupies.
size – The size of the color component arrays.
r – The array of red color components.
g – The array of green color components.
b – The array of blue color components.
a – The array of alpha value components.

o **IndexColorModel**

```
public IndexColorModel(int bits,
                       int size,
                       byte cmap[],
                       int start,
                       boolean hasalpha)
```

Constructs an IndexColorModel from a single array of packed red, green, blue and optional alpha components. The array must have enough values in it to fill all of the needed component arrays of the specified size. Uses the default sRGB ColorSpace.

Parameters:

bits – The number of bits each pixel occupies.
size – The size of the color component arrays.
cmap – The array of color components.
start – The starting offset of the first color component.
hasalpha – Indicates whether alpha values are contained in the cmap array.

o **IndexColorModel**

```
public IndexColorModel(int bits,
                       int size,
                       byte cmap[],
                       int start,
```

```
                        boolean hasalpha,
                        int trans)
```

Constructs an IndexColorModel from a single array of packed red, green, blue and optional alpha components. The specified transparent index represents a pixel that will be considered entirely transparent regardless of any alpha value specified for it. The array must have enough values in it to fill all of the needed component arrays of the specified size. Uses the default sRGB ColorSpace.

Parameters:
- bits – The number of bits each pixel occupies.
- size – The size of the color component arrays.
- cmap – The array of color components.
- start – The starting offset of the first color component.
- hasalpha – Indicates whether alpha values are contained in the cmap array.
- trans – The index of the fully transparent pixel.

o **IndexColorModel**

```
public IndexColorModel(int bits,
                        int size,
                        int cmap[],
                        int start,
                        boolean hasalpha,
                        int trans)
```

Constructs an IndexColorModel from an array of integers where each integer is comprised of red, green, blue, and optional alpha components in the default RGB color model packing. The specified transparent index represents a pixel that will be considered entirely transparent regardless of any alpha value specified for it. The array must have enough values in it to fill all of the needed component arrays of the specified size. Uses the default sRGB ColorSpace.

Parameters:
- bits – The number of bits each pixel occupies.
- size – The size of the color component arrays.
- cmap – The array of color components.
- start – The starting offset of the first color component.
- hasalpha – Indicates whether alpha values are contained in the cmap array.
- trans – The index of the fully transparent pixel.

Methods

o **getTransparency**

```
public int getTransparency()
```

Returns the transparency. Returns either OPAQUE, BITMASK, or TRANSLUCENT

Overrides:

getTransparency in class ColorModel

See Also:

OPAQUE, BITMASK, TRANSLUCENT

o **getComponentSize**

```
public int[] getComponentSize()
```

Returns an array of the number of bits per component.

Overrides:

getComponentSize in class ColorModel

o **getMapSize**

```
public final int getMapSize()
```

Returns the size of the color component arrays in this IndexColorModel.

o **getTransparentPixel**

```
public final int getTransparentPixel()
```

Returns the index of the transparent pixel in this IndexColorModel or −1 if there is no transparent pixel.

o **getReds**

```
public final void getReds(byte r[])
```

Copies the array of red color components into the given array. Only the initial entries of the array as specified by getMapSize() are written. If the color space is not sRGB, then the entries are converted to sRGB and the red components are extracted.

o **getGreens**

```
public final void getGreens(byte g[])
```

Copies the array of green color components into the given array. Only the initial entries of the array as specified by getMapSize() are written. If the color space is not sRGB, then the entries are converted to sRGB and the green components are extracted.

o **getBlues**

```
public final void getBlues(byte b[])
```

Copies the array of blue color components into the given array. Only the initial entries of the array as specified by getMapSize() will be written. If the color space is not sRGB, then the entries are converted to sRGB and the green components are extracted.

o **getAlphas**

```
public final void getAlphas(byte a[])
```

Copies the array of alpha transparency values into the given array. Only the initial entries of the array as specified by getMapSize() will be written.

o **getRGBs**

```
public final void getRGBs(int rgb[])
```

Converts the arrays color and alpha components to the default RGB ColorModel (if necessary) and copies the resulting array of 32−bit ARGB values into the given array. Only the initial entries of the array as specified by getMapSize() will be written.

o **getRed**

```
public final int getRed(int pixel)
```

Returns the red color component for the specified pixel in the range 0−255 in the default RGB ColorSpace, sRGB. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getRed in class ColorModel

o **getGreen**

```
public final int getGreen(int pixel)
```

Returns the green color component for the specified pixel in the range 0−255 in the default RGB ColorSpace, sRGB. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getGreen in class ColorModel

o **getBlue**

```
public final int getBlue(int pixel)
```

Returns the blue color component for the specified pixel in the range 0−255 in the default RGB ColorSpace, sRGB. The returned value will be a non pre−multiplied color (i.e. if the alpha is premultiplied, this method will divide it out before it is returned). If the alpha value is 0, the color will be 0.

Overrides:
getBlue in class ColorModel

o **getAlpha**

```
public final int getAlpha(int pixel)
```

Return the alpha transparency value for the specified pixel in the range 0–255.

Overrides:
getAlpha in class ColorModel

o **getRGB**

```
public final int getRGB(int pixel)
```

Returns the color of the pixel in the default RGB color model. If the ColorSpace is not sRGB, it will be converted to sRGB.

Overrides:
getRGB in class ColorModel
See Also:
getRGBdefault

o **getPixel**

```
public int[] getPixel(int rgb,
                      int pixel[])
```

Returns a pixel array given a pixel in the default RGB color model. This pixel array can then be passed to the putData method of a Tile object. If the pixel variable is null, a new array will be allocated. The pixel array will be returned.

Overrides:
getPixel in class ColorModel

o **getComponents**

```
public int[] getComponents(int pixel,
                           int components[])
```

Returns an array of all of the components given a pixel in the current color model. This can be used to retrieve the components for a specific pixel. If the components array is null, a new array will be allocated. The components array will be returned.

Overrides:
getComponents in class ColorModel

o **getComponents**

```
public int[] getComponents(int pixel[],
                           int components[])
```

Returns an array of all of the components given a pixel array in the current color model. This can be used to retrieve the components for a specific pixel in a PackedColorModel. If the components array is null, a new array will be allocated. The components array will be returned.

Overrides:
getComponents in class ColorModel

o **getDataArray**

```
public void getDataArray(int n,
                         byte data_array[])
```

Copies the nth data array (color or alpha component) into the given array. Only the initial entries of the array as specified by getMapSize() will be written.

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a Tile with the specified width and height, that has a channel layout compatible with this ColorModel.

Overrides:
createCompatibleTile in class ColorModel

See Also:
Tile

o **coerceData**

```
public ColorModel coerceData(Tile tile,
                             boolean isAlphaPremultiplied)
```

Forces the Tile data to match the state specified in the isAlphaPremultiplied variable, assuming the data is currently correctly described by this ColorModel. It may multiply or divide the color Tile data by alpha, or do nothing if the data is in the correct state. If the data needs to be coerced, this method will also return an instance of this ColorModel with the isAlphaPremultiplied flag set appropriately.

Overrides:
coerceData in class ColorModel

o **convertToIntDiscrete**

```
public BufferedImage convertToIntDiscrete(Tile tile,
                                          boolean forceARGB)
```

Returns a new BufferedImage of TYPE_INT_ARGB or TYPE_INT_RGB that matches the lookup tables.

Class java.awt.image.IntegerComponentTile

```
java.lang.Object
   |
   +----java.awt.image.Tile
           |
           +----java.awt.image.IntegerComponentTile
```

---

public class **IntegerComponentTile**
extends Tile

This class defines a tile with pixels consisting of one or more 32−bit channels stored in close proximity to each other in a integer array. The bit precision per channel element is that of the data type (that is, the bit precision for this tile is 32). There is only one pixel stride and one scanline stride for all channels. This type of tile can be used with a ComponentColorModel if there are multiple channels, or a PackedColorModel (including a DirectColorModel) if there is only one channel.

The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Constructor Index

o **IntegerComponentTile**(int, int, int, int, int, int, int[])
  Constructs a base tile with 1 channel and a pixel stride of 1.
o **IntegerComponentTile**(int, int, int, int, int, int[], int, int, int[])
  Constructs a base tile given the channel array and tile dimensions.
o **IntegerComponentTile**(int, int, int, int[], int, int, int[])
  Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
  Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.
o **createCompatibleTile**(int, int)

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.

o **createSubtile**(int, int, int, int)
  Creates a subtile given a region of the tile.

o **createSubtile**(int, int, int, int, int[])
  Creates a subtile given a region of the tile.

o **getChannelOffset**(int)
  Returns channel offset for the specified channel.

o **getChannelOffsets**()
  Returns a copy of the channel offsets array.

o **getData**(int, int, int)
  Returns the channel element for the specified channel at the specified location.

o **getData**(int, int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.

o **getData**(int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.

o **getData**(int, int, int[])
  Returns the channel elements for all channels at the specified location.

o **getDataStorage**()
  Returns a reference to the entire data array

o **getPixelStride**()
  Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.

o **getScanlineStride**()
  Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **print**(int, int, int, int)
  For debugging...

o **putData**(int, int, int, int)
  Stores the channel element for the specified channel at the specified location.

o **putData**(int, int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.

o **putData**(int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.

o **putData**(int, int, int[])
  Stores the channel elements for all channels at the specified location.

o **putData**(int, int, Tile)
  Stores the Tile data at the specified location.

o **toString**()

Constructors

o **IntegerComponentTile**

```
public IntegerComponentTile(int width,
                            int height,
                            int numChannels,
                            int channelOffsets[],
                            int pixelStride,
```

```
                              int scanlineStride,
                              int data[])
```

Constructs a base tile given the channel array and tile dimensions. The tile data will start at x=0, y=0 in the data array of each channel.

Parameters:

width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException

if the channel parameters are incompatible with the tile width and height.

o **IntegerComponentTile**

```
public IntegerComponentTile(int x,
                              int y,
                              int width,
                              int height,
                              int numChannels,
                              int channelOffsets[],
                              int pixelStride,
                              int scanlineStride,
                              int data[])
```

Constructs a base tile given the channel array and tile dimensions.

Parameters:

x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException

if the channel parameters are incompatible with the tile x offset, y offset, width and height.

o **IntegerComponentTile**

```
public IntegerComponentTile(int x,
                            int y,
                            int width,
                            int height,
                            int channelOffset,
                            int scanlineStride,
                            int data[])
```

Constructs a base tile with 1 channel and a pixel stride of 1.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
channelOffset – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getChannelOffsets**

```
public int[] getChannelOffsets()
```

Returns a copy of the channel offsets array.

o **getChannelOffset**

```
public int getChannelOffset(int channel)
```

Returns channel offset for the specified channel. The channel offset is the index into the channel data array in which the first channel element of the first scanline is stored.

o **getScanlineStride**

```
public int getScanlineStride()
```

Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **getPixelStride**

```
public int getPixelStride()
```

Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.

o **getDataStorage**

```
public int[] getDataStorage()
```

Returns a reference to the entire data array

o **getData**

```
public int getData(int x,
                   int y,
                   int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int outData[])
```

Returns the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
outData – If non–null, channel elements for all channels at the specified location are returned in this array.
Returns:
Data array with channel elements for all channels.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int channel,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
int[] pixel = new int[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

Overrides:

getData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int channel,
                    int inData)
```

Stores the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown

at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
inData – Channel element.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inData – Channel elements for all channels.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayIndexOutOfBoundsException will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inTile – Tile of data to place at x,y location.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
```

```
                  int channel,
                  int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x − X offset.
y − Y offset.
width − Width (in pixels) of the subtile

height – Height (in pixels) of the subtile

Throws: TileFormatException

if the specified bounding box is outside of the parent tile.

Overrides:

createSubtile in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height,
                          int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper–left corner of this tile to the upper–left corner of the subtile. A subset of the channels of the parent tile may be specified. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:

x – X offset.
y – Y offset.
width – Width (in pixels) of the subtile
height – Height (in pixels) of the subtile
channelList – Array of channel indices.

Throws: TileFormatException

if the specified bounding box is outside of the parent tile.

Overrides:

createSubtile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Parameters that are guaranteed to remain the same are:

- number of channels
- pixel stride

Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
        may not be equal to:
gc.createCompatibleTile(w2, h2)
```

Overrides:

createCompatibleTile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays. If the tile is a subtile, this will call createCompatibleTile(width, height).

Overrides:
createCompatibleTile in class Tile

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

o **print**

```
public void print(int x,
                  int y,
                  int w,
                  int h)
```

For debugging... prints a region of a one−band IntegerComponentTile

---

Class java.awt.image.Kernel

```
java.lang.Object
   |
   +----java.awt.image.Kernel
```

---

public class **Kernel**
extends Object

This class defines a Kernel object. A kernel is a matrix describing how a given pixel and its surrounding pixels affect the value of the given pixel in a filtering operation.

See Also:
ConvolveOp

Variable Index

o **SHARPEN3x3_1**
3x3 Sharpening kernel:

```
 -1 -1 -1
 -1  9 -1
 -1 -1 -1
```

o **SHARPEN3x3_2**
3x3 Sharpening kernel:

```
  1 -2  1
 -2  5 -2
  1 -2  1
```

o **SHARPEN3x3_3**
3x3 Sharpening kernel:

```
  0 -1  0
 -1  5 -1
  0 -1  0
```

Constructor Index

o **Kernel**(int, int, int, int, float[])
  Create a Kernel object from an array of floats.

Method Index

o **getHeight**()
  Returns the height
o **getKernelData**(float[])
  Returns the kernel data in row major order.
o **getWidth**()
  Returns the width
o **getXOrigin**()
  Returns the X origin.
o **getYOrigin**()
  Returns the Y origin.

Variables

o **SHARPEN3x3_1**

```
public static final float SHARPEN3x3_1[]
```

3x3 Sharpening kernel:

```
-1 -1 -1
-1  9 -1
-1 -1 -1
```

o **SHARPEN3x3_2**

```
public static final float SHARPEN3x3_2[]
```

3x3 Sharpening kernel:

```
 1 -2  1
-2  5 -2
 1 -2  1
```

o **SHARPEN3x3_3**

```
public static final float SHARPEN3x3_3[]
```

3x3 Sharpening kernel:

```
 0 -1  0
-1  5 -1
 0 -1  0
```

Constructors

o **Kernel**

```
public Kernel(int width,
              int height,
              int xOrigin,
              int yOrigin,
              float data[])
```

Create a Kernel object from an array of floats. The data array is copied.

Parameters:
width − Width of the kernel.
height − Height of the kernel.
xOrigin − X origin of the kernel (usually the center) with respect to the upper−left corner.
yOrigin − Y origin of the kernel (usually the center) with respect to the upper−left corner.
data − Kernel data in row major order.

Methods

o **getXOrigin**

```
public int getXOrigin()
```

Returns the X origin.

o **getYOrigin**

```
public int getYOrigin()
```

Returns the Y origin.

o **getWidth**

```
public int getWidth()
```

Returns the width

o **getHeight**

```
public int getHeight()
```

Returns the height

o **getKernelData**

```
public float[] getKernelData(float data[])
```

Returns the kernel data in row major order.

Parameters:
data − If non−null, will contain the returned kernel data.

Class java.awt.image.LookupOp

```
java.lang.Object
   |
   +----java.awt.image.LookupOp
```

public class **LookupOp**
extends Object
implements BufferedImageOp, TileOp

This class implements a lookup operation from the source to the destination. The LookupTable object may contain a single array or multiple arrays, subject to the restrictions below.

For Tiles, the lookup operates on channel elements. The number of lookup arrays may be one, in which case the same array is applied to all channels, or it must equal the number of Source tile channels.

For BufferedImages, the lookup operates on color and alpha components. The number of lookup arrays may be one, in which case the same array is applied to all color and alpha components, or it must equal the number of Source color components, in which case no lookup of the alpha component (if present) is performed, or it must equal the number of Source color components plus alpha components, in which case lookup is performed for all color and alpha components. This allows non−uniform rescaling of multi−channelled BufferedImages.

This class allows the Source to be the same as Destination.

See Also:
LookupTable

Constructor Index

o **LookupOp**(LookupTable)
Constructs a LookupOp object given the lookup table.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
    Creates an empty destination image with the correct size and number of channels.
o **createCompatibleDestTile**(Tile)
    Creates an empty destination tile with the correct size and number of channels.
o **filter**(BufferedImage, BufferedImage)
    Performs a lookup operation on a BufferedImage.
o **filter**(Tile, Tile)
    Performs a lookup operation on a Tile.
o **getDestBounds**(BufferedImage)
    Returns the bounding box of the destination.
o **getDestBounds**(Tile)
    Returns the bounding box of the destination.
o **getDestPoint**(Point2D, Point2D)
    Returns the location of the destination point given a point in the source image.
o **getLookupTable**()
    Returns the LookupTable.

Constructors

o **LookupOp**

```
public LookupOp(LookupTable lookup)
```

Constructs a LookupOp object given the lookup table.

Methods

o **getLookupTable**

```
public LookupTable getLookupTable()
```

Returns the LookupTable.

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst)
```

Performs a lookup operation on a BufferedImage. If the color model in the source image is not the same as that in the destination image, the pixels will be converted in the destination. If the destination image is null, a BufferedImage will be created with the source ColorModel. The IllegalArgumentException may be thrown if the number of arrays in the LookupTable does not meet the restrictions stated in the class comment above.

o **filter**

```
public Tile filter(Tile src,
                   Tile dst)
```

Performs a lookup operation on a Tile. If the destination Tile is null, a new Tile will be created. The IllegalArgumentException may be thrown if the source and destination tiles have different number of channels or if the number of arrays in the LookupTable does not meet the restrictions stated in the class comment above.

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the destination. Since this is not a geometric operation, the bounding box does not change. The IllegalArgumentException may be thrown if the number of arrays in the LookupTable does not meet the restrictions stated in the class comment above.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the destination. Since this is not a geometric operation, the bounding box does not change. The IllegalArgumentException may be thrown if the number of arrays in the LookupTable does not meet the restrictions stated in the class comment above.

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of channels. The IllegalArgumentException may be thrown if the number of arrays in the LookupTable does not meet the restrictions stated in the class comment above.

Parameters:
src – Source image for the filter operation.
destCM – ColorModel of the destination. If null, the ColorModel of the source will be used.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels. The IllegalArgumentException may be thrown if the number of arrays in the LookupTable does not meet the restrictions stated in the class comment above.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                             Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non−null, it will be used to hold the return value. Since this is not a geometric operation, the srcPt will equal the dstPt.

Class java.awt.image.NearestNeighborAffineTransformOp

```
java.lang.Object
   |
   +----java.awt.image.AffineTransformOp
           |
           +----java.awt.image.NearestNeighborAffineTransformOp
```

public class **NearestNeighborAffineTransformOp**
extends AffineTransformOp

This class uses an affine transformation with nearest neighbor interpolation to transform an image or tile.

Note that the following constraints have to be met:

- Source and Destination must be different.
- For Tiles, the number of Channels in the Source must be equal to the number of Channels in the Destination.

Constructor Index

o **NearestNeighborAffineTransformOp**(AffineTransform)
Constructs a NearestNeighborAffineTransformOp given an affine transform.

Method Index

o **filter**(BufferedImage, BufferedImage)
Transforms the BufferedImage using the stored affine transform.
o **filter**(Tile, Tile)
Transforms the Tile using the stored affine transform.

Constructors

o **NearestNeighborAffineTransformOp**

```
public NearestNeighborAffineTransformOp(AffineTransform xform)
```

Constructs a NearestNeighborAffineTransformOp given an affine transform.

Methods

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst)
```

Transforms the BufferedImage using the stored affine transform. If the color models for the two images do not match, a color conversion into the destination color model will be performed. If the destination image is null, a BufferedImage will be created with the source ColorModel. The IllegalArgumentException may be thrown if the source is the same as the destination.

Overrides:
filter in class AffineTransformOp

o **filter**

```
public Tile filter(Tile src,
                   Tile dest)
```

Transforms the Tile using the stored affine transform. This operation will perform the transform channel by channel. If the destination Tile is null, a new Tile will be created. The IllegalArgumentException may be thrown if the source is the same as the destination or if the number of channels in the source is not equal to the number of channels in the destination.

Overrides:
filter in class AffineTransformOp

Class java.awt.image.PackedColorModel

```
java.lang.Object
   |
   +----java.awt.image.ColorModel
           |
           +----java.awt.image.PackedColorModel
```

public abstract class **PackedColorModel**
extends ColorModel
super class of:
    DirectColorModel

An abstract ColorModel class that represents pixel values that have the color components embedded directly in the bits of an integer pixel. A subclass is the DirectColorModel, which is similar to an X11 TrueColor visual.

See Also:
    DirectColorModel

Constructor Index

o **PackedColorModel**(ColorSpace, int, int, int, int, int, boolean, int)
    Constructs a PackedColorModel from the given masks specifying which bits in the pixel contain the alpha, red, green and blue color components.
o **PackedColorModel**(ColorSpace, int, int[], int, boolean, int)
    Constructs a PackedColorModel from a mask array, which specifies which bits in the integer pixel contain each of the color components in the ColorSpace.

Method Index

o **equals**(Object)
o **getMask**(int)
    Returns the mask indicating which bits in a pixel contain the color component.

Constructors

o **PackedColorModel**

```
public PackedColorModel(ColorSpace space,
                        int bits,
                        int colorMaskArray[],
                        int alphaMask,
                        boolean isAlphaPremultiplied,
                        int trans)
```

Constructs a PackedColorModel from a mask array, which specifies which bits in the integer pixel contain each of the color components in the ColorSpace. All of the bits in each mask must be contiguous and fit in the specified number of least significant bits of the integer. If the alphaMask is 0, there is no alpha. If there is alpha, the boolean, isAlphaPremultiplied, specifies how to interpret color data that are passed to this ColorModel. If it is true, the color data will have been multiplied with alpha.

o **PackedColorModel**

```
public PackedColorModel(ColorSpace space,
                        int bits,
                        int rmask,
                        int gmask,
                        int bmask,
                        int amask,
                        boolean isAlphaPremultiplied,
                        int trans)
```

Constructs a PackedColorModel from the given masks specifying which bits in the pixel contain the alpha, red, green and blue color components. All of the bits in each mask must be contiguous and fit in the specified number of least significant bits of the integer. The boolean, isAlphaPremultiplied, specifies how to interpret color data that are passed to this ColorModel. If it is true, the color data will have been multiplied with alpha. The ColorSpace should be of type ColorSpace.TYPE_RGB.

See Also:
ColorSpace

Methods

o **getMask**

```
public int getMask(int index)
```

Returns the mask indicating which bits in a pixel contain the color component. The index corresponds to the placement of color component names in the color space. Thus, index 0 for a CMYK ColorSpace would correspond to Cyan and index 1 would correspond to Magenta. If there is alpha, the alpha index would be:

```
alphaIndex = numComponents() - 1;
```

o **equals**

```
public boolean equals(Object obj)
```

Overrides:
equals in class ColorModel

Class java.awt.image.RescaleOp

```
java.lang.Object
   |
   +----java.awt.image.RescaleOp
```

---

public class **RescaleOp**
extends Object
implements BufferedImageOp, TileOp

This class 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. The pixel values are clipped to the minimum/maximum of the destination image data type.

For Tiles, rescaling operates on channel elements. The number of sets of scaling constants may be one, in which case the same constants are applied to all channels, or it must equal the number of Source tile channels.

For BufferedImages, rescaling operates on color and alpha components. The number of sets of scaling constants may be one, in which case the same constants are applied to all color and alpha components, or it must equal the number of Source color components, in which case no scaling of the alpha component (if present) is performed, or it must equal the number of Source color components plus alpha components, in which case all color and alpha components are scaled.

The pseudo code for the rescaling operation is as follows:

```
for each pixel from Source object {
    for each channel/component of the pixel {
        dstElement = (srcElement*scaleFactor) + offset
    }
}
```

Note that in−place operation is allowed (i.e. the source and destination can be the same object).

Constructor Index

o **RescaleOp**(float, float)

Constructs a new RescaleOp with the desired scale factor and offset.

o **RescaleOp**(float[], float[])

Constructs a new RescaleOp with the desired scale factors and offsets.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)

Creates an empty destination image with the correct size and number of channels.

o **createCompatibleDestTile**(Tile)

Creates an empty destination tile with the correct size and number of channels.

o **filter**(BufferedImage, BufferedImage)

Rescales the BufferedImage.

o **filter**(Tile, Tile)

Rescales the channel data in the Tile.

o **getDestBounds**(BufferedImage)

Returns the bounding box of the rescaled destination.

o **getDestBounds**(Tile)

Returns the bounding box of the rescaled destination.

o **getDestPoint**(Point2D, Point2D)

Returns the location of the destination point given a point in the source image.

o **getNumFactors**()

Returns the number of scaling factors used in this object.

o **getOffsets**(float[])

Returns the offsets in the given array.

o **getScaleFactors**(float[])

Returns the scale factors in the given array.

Constructors

o **RescaleOp**

```
public RescaleOp(float scaleFactors[],
                 float offsets[])
```

Constructs a new RescaleOp with the desired scale factors and offsets. The length of the scaleFactor and offset arrays must meet the restrictions stated in the class comments above.

o **RescaleOp**

```
public RescaleOp(float scaleFactor,
                 float offset)
```

Constructs a new RescaleOp with the desired scale factor and offset. The scaleFactor and offset will be applied to all channels/components in the source Tile/BufferedImage.

Methods

o **getScaleFactors**

```
public float[] getScaleFactors(float scaleFactors[])
```

Returns the scale factors in the given array. The array is also returned for convenience.

o **getOffsets**

```
public float[] getOffsets(float offsets[])
```

Returns the offsets in the given array. The array is also returned for convenience.

o **getNumFactors**

```
public int getNumFactors()
```

Returns the number of scaling factors used in this object.

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dst)
```

Rescales the BufferedImage. If the color model in the source image is not the same as that in the destination image, the pixels will be converted in the destination. If the destination image is null, a BufferedImage will be created with the source ColorModel. Note that the number of scaling factors in this object must meet the restrictions stated in the class comments above. Otherwise, an exception is thrown.

o **filter**

```
public Tile filter(Tile src,
                   Tile dest)
```

Rescales the channel data in the Tile. If the destination Tile is null, a new Tile will be created. The source and destination must have the same number of channels. Otherwise, an exception is thrown. Note that the number of scaling factors in this object must meet the restrictions stated in the class comments above. Otherwise, an exception is thrown.

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the rescaled destination. Since this is not a geometric operation, the bounding box does not change. Note that the number of scaling factors in this object must meet the restrictions stated in the class comments above. Otherwise, an exception is thrown.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the rescaled destination. Since this is not a geometric operation, the bounding box does not change. Note that the number of scaling factors in this object must meet the restrictions stated in the class comments above. Otherwise, an exception is thrown.

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of channels. Note that the number of scaling factors in this object must meet the restrictions stated in the class comments above. Otherwise, an exception is thrown.

Parameters:
src – Source image for the filter operation.
destCM – ColorModel of the destination. If null, the ColorModel of the source will be used.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels. Note that the number of scaling factors in this object must meet the restrictions stated in the class comments above. Otherwise, an exception is thrown.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                            Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non–null, it will be used to hold the return value. Since this is not a geometric operation, the srcPt will equal the dstPt.

Class java.awt.image.ShortBandedTile

```
java.lang.Object
   |
   +----java.awt.image.Tile
           |
           +----java.awt.image.ShortBandedTile
```

---

public class **ShortBandedTile**
extends Tile

This class defines a tile with pixels consisting of multiple 16−bit component stored in separate arrays for each component. Operations on sets of pixels are performed on a given component of each pixel in the set before moving on to the next component. The arrays used for storage may be distinct or shared between some or all of the components. There is only one pixel stride and one scanline stride for all channels. This type of tile can be used with a ComponentColorModel.

The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Constructor Index

o **ShortBandedTile**(int, int, int, int, int, int[], int, int, short[][])
    Constructs a base tile given the tile parameters.
o **ShortBandedTile**(int, int, int, int[], int, int, short[][])
    Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
    Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.
o **createCompatibleTile**(int, int)
    Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.

o **createSubtile**(int, int, int, int)
    Creates a subtile given a region of the tile.
o **createSubtile**(int, int, int, int, int[])
    Creates a subtile given a region of the tile.
o **getChannelOffset**(int)
    Returns channel offset for the specified channel.
o **getChannelOffsets**()
    Returns a copy of the channel offsets array.
o **getData**(int, int, int)
    Returns the channel element for the specified channel at the specified location.
o **getData**(int, int, int, int, int, int[])
    Returns an integer array of pixels from the specified rectangular region.
o **getData**(int, int, int, int, int[])
    Returns an integer array of pixels from the specified rectangular region.
o **getData**(int, int, int[])
    Returns the channel elements for all channels at the specified location.
o **getDataStorage**()
    Returns a reference to the entire data array
o **getDataStorage**(int)
    Returns a reference to the specific channel data array
o **getPixelStride**()
    Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.
o **getScanlineStride**()
    Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.
o **getShortData**(int, int, int, int, int, short[])
    Returns a short integer array of pixels from the specified rectangular region.
o **getShortData**(int, int, int, int, short[])
    Returns a short integer array of pixels from the specified rectangular region.
o **putData**(int, int, int, int)
    Stores the channel element for the specified channel at the specified location.
o **putData**(int, int, int, int, int, int[])
    Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int, int, int[])
    Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int[])
    Stores the channel elements for all channels at the specified location.
o **putData**(int, int, Tile)
    Stores the Tile data at the specified location.
o **putShortData**(int, int, int, int, int, short[])
    Stores a short integer array of pixels into the specified rectangular region.
o **putShortData**(int, int, int, int, short[])
    Stores a short integer array of pixels into the specified rectangular region.
o **toString**()

Constructors

o **ShortBandedTile**

```
public ShortBandedTile(int width,
                       int height,
                       int numChannels,
                       int channelOffsets[],
                       int pixelStride,
                       int scanlineStride,
                       short data[][])
```

Constructs a base tile given the channel array and tile dimensions. The tile data will start at x=0, y=0 in the data array of each channel.

Parameters:
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile width and height.

o **ShortBandedTile**

```
public ShortBandedTile(int x,
                       int y,
                       int width,
                       int height,
                       int numChannels,
                       int channelOffsets[],
                       int pixelStride,
                       int scanlineStride,
                       short data[][])
```

Constructs a base tile given the tile parameters.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffset – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.

pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getChannelOffsets**

```
public int[] getChannelOffsets()
```

Returns a copy of the channel offsets array.

o **getChannelOffset**

```
public int getChannelOffset(int channel)
```

Returns channel offset for the specified channel. The channel offset is the index into the channel data array in which the first channel element of the first scanline is stored.

o **getScanlineStride**

```
public int getScanlineStride()
```

Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **getPixelStride**

```
public int getPixelStride()
```

Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.

o **getDataStorage**

```
public short[][] getDataStorage()
```

Returns a reference to the entire data array

o **getDataStorage**

```
public short[] getDataStorage(int channel)
```

Returns a reference to the specific channel data array

o **getData**

```
public int getData(int x,
                   int y,
                   int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int outData[])
```

Returns the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
outData – If non–null, channel elements for all channels at the specified location are returned in this array.
Returns:
Data array with channel elements for all channels.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int channel,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown

at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
int[] pixel = new int[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

Overrides:

getData in class Tile

o **getShortData**

```
public short[] getShortData(int x,
                            int y,
                            int w,
                            int h,
                            int channel,
                            short outData[])
```

Returns a short integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
short[] channelData = tile.getShortData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
short chanElement = channelData[((y2-y)*w + (x2-x))];
```

o **getShortData**

```
public short[] getShortData(int x,
                            int y,
                            int w,
                            int h,
                            short outData[])
```

Returns a short integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
short[] channelData = tile.getShortData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
short[] pixel = new short[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

o **putData**

```
public void putData(int x,
                    int y,
                    int channel,
                    int inData)
```

Stores the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
inData – Channel element.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inData – Channel elements for all channels.
Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inTile – Tile of data to place at x,y location.
Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int channel,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are

assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putShortData**

```
public void putShortData(int x,
                         int y,
                         int w,
                         int h,
                         int channel,
                         short inData[])
```

Stores a short integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))]
```

o **putShortData**

```
public void putShortData(int x,
                         int y,
                         int w,
                         int h,
                         short inData[])
```

Stores a short integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile.

Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x – X offset.
y – Y offset.
width – Width (in pixels) of the subtile
height – Height (in pixels) of the subtile

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height,
                          int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper–left corner of this tile to the upper–left corner of the subtile. A subset of the parent tile's channels may be specified. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
x – X offset.
y – Y offset.
width – Width (in pixels) of the subtile
height – Height (in pixels) of the subtile
channelList – Array of channel indices.

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Parameters that are guaranteed to remain the same are:

- number of channels
- pixel stride

Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
        may not be equal to:
```

```
gc.createCompatibleTile(w2, h2)
```

Overrides:
createCompatibleTile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays. If the tile is a subtile, this will call createCompatibleTile(width, height).

Overrides:
createCompatibleTile in class Tile

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

Class java.awt.image.ShortComponentTile

```
java.lang.Object
   |
   +----java.awt.image.Tile
           |
           +----java.awt.image.ShortComponentTile
```

---

public class **ShortComponentTile**
extends Tile

This class defines a tile with pixels consisting of one or more 16–bit channels stored in close proximity to each other in a short integer array. The bit precision per channel element is that of the data type (that is, the bit precision for this tile is 16). There is only one pixel stride and one scanline stride for all channels. This type of tile can be used with a ComponentColorModel if there are multiple channels, or a IndexColorModel if there is only one channel.

The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Constructor Index

o **ShortComponentTile**(int, int, int, int, int, int, short[])
    Constructs a base tile with 1 channel and a pixel stride of 1.
o **ShortComponentTile**(int, int, int, int, int, int[], int, int, short[])
    Constructs a base tile given the channel array and tile dimensions.
o **ShortComponentTile**(int, int, int, int[], int, int, short[])
    Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
    Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.
o **createCompatibleTile**(int, int)

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.

o **createSubtile**(int, int, int, int)
  Creates a subtile given a region of the tile.
o **createSubtile**(int, int, int, int, int[])
  Creates a subtile given a region of the tile.
o **getChannelOffset**(int)
  Returns channel offset for the specified channel.
o **getChannelOffsets**()
  Returns a copy of the channel offsets array.
o **getData**(int, int, int)
  Returns the channel element for the specified channel at the specified location.
o **getData**(int, int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.
o **getData**(int, int, int, int, int[])
  Returns an integer array of pixels from the specified rectangular region.
o **getData**(int, int, int[])
  Returns the channel elements for all channels at the specified location.
o **getDataStorage**()
  Returns a reference to the entire data array
o **getPixelStride**()
  Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.
o **getScanlineStride**()
  Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.
o **getShortData**(int, int, int, int, int, short[])
  Returns a short integer array of pixels from the specified rectangular region.
o **getShortData**(int, int, int, int, short[])
  Returns a short integer array of pixels from the specified rectangular region.
o **putData**(int, int, int, int)
  Stores the channel element for the specified channel at the specified location.
o **putData**(int, int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int, int, int[])
  Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int[])
  Stores the channel elements for all channels at the specified location.
o **putData**(int, int, Tile)
  Stores the Tile data at the specified location.
o **putShortData**(int, int, int, int, int, short[])
  Stores a short integer array of pixels into the specified rectangular region.
o **putShortData**(int, int, int, int, short[])
  Stores a short integer array of pixels into the specified rectangular region.
o **toString**()

Constructors

o **ShortComponentTile**

```
public ShortComponentTile(int width,
                          int height,
                          int numChannels,
                          int channelOffsets[],
                          int pixelStride,
                          int scanlineStride,
                          short data[])
```

Constructs a base tile given the channel array and tile dimensions. The tile data will start at x=0, y=0 in the data array of each channel.

Parameters:
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile width and height.

o **ShortComponentTile**

```
public ShortComponentTile(int x,
                          int y,
                          int width,
                          int height,
                          int numChannels,
                          int channelOffsets[],
                          int pixelStride,
                          int scanlineStride,
                          short data[])
```

Constructs a base tile given the channel array and tile dimensions.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
numChannels – Number of channels in this tile.
channelOffsets – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.

pixelStride – Number of data array elements between two channel elements for the same channel on the same scanline.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

o **ShortComponentTile**

```
public ShortComponentTile(int x,
                          int y,
                          int width,
                          int height,
                          int channelOffset,
                          int scanlineStride,
                          short data[])
```

Constructs a base tile with 1 channel and a pixel stride of 1.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
channelOffset – Index into the channel data array in which the first channel element of the first scanline of this channel is stored.
scanlineStride – Number of data array elements between a given channel element and the channel element in the same column of the next row.
data – The data array.

Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getChannelOffsets**

```
public int[] getChannelOffsets()
```

Returns a copy of the channel offsets array.

o **getChannelOffset**

```
public int getChannelOffset(int channel)
```

Returns channel offset for the specified channel. The channel offset is the index into the channel data array in which the first channel element of the first scanline is stored.

o **getScanlineStride**

```
public int getScanlineStride()
```

Returns the scanline stride –– the number of data array elements between a given channel element and the channel element in the same column of the next row.

o **getPixelStride**

```
public int getPixelStride()
```

Returns pixel stride –– the number of data array elements between two channel elements for the same channel on the same scanline.

o **getDataStorage**

```
public short[] getDataStorage()
```

Returns a reference to the entire data array

o **getData**

```
public int getData(int x,
                   int y,
                   int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
Overrides:
getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int outData[])
```

Returns the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.

y – Y coordinate.
outData – If non–null, channel elements for all channels at the specified location are returned in this array.

Returns:

Data array with channel elements for all channels.

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int channel,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

Overrides:

getData in class Tile

o **getData**

```
public int[] getData(int x,
                     int y,
                     int w,
                     int h,
                     int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
int[] pixel = new int[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

Overrides:

getData in class Tile

o **getShortData**

```
public short[] getShortData(int x,
                            int y,
                            int w,
                            int h,
                            int channel,
                            short outData[])
```

Returns a short integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
short[] channelData = tile.getShortData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
short chanElement = channelData[((y2-y)*w + (x2-x))];
```

o **getShortData**

```
public short[] getShortData(int x,
                            int y,
                            int w,
                            int h,
                            short outData[])
```

Returns a short integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
short[] channelData = tile.getShortData(x, y, w, h, null);
int numChannels = tile.getNumChannels();
short[] pixel = new short[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

o **putData**

```
public void putData(int x,
                    int y,
                    int channel,
                    int inData)
```

Stores the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.
y – Y coordinate.
channel – Channel index.

inData – Channel element.

Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.
y – Y coordinate.
inData – Channel elements for all channels.

Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:

x – X coordinate.
y – Y coordinate.
inTile – Tile of data to place at x,y location.

Overrides:

putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int channel,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are

assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putData**

```
public void putData(int x,
                    int y,
                    int w,
                    int h,
                    int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

Overrides:
putData in class Tile

o **putShortData**

```
public void putShortData(int x,
                         int y,
                         int w,
                         int h,
                         int channel,
                         short inData[])
```

Stores a short integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))]
```

o **putShortData**

```
public void putShortData(int x,
                         int y,
                         int w,
                         int h,
```

```
                          short inData[])
```

Stores a short integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
- x − X offset.
- y − Y offset.
- width − Width (in pixels) of the subtile
- height − Height (in pixels) of the subtile

Throws: TileFormatException
- if the specified bounding box is outside of the parent tile.

Overrides:
- createSubtile in class Tile

o **createSubtile**

```
public Tile createSubtile(int x,
                          int y,
                          int width,
                          int height,
                          int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper−left corner of this tile to the upper−left corner of the subtile. A subset of the channels of the parent tile may be specified. Note that the subtile will reference the same channel objects as the parent tile, but using different offsets.

Parameters:
- x − X offset.
- y − Y offset.
- width − Width (in pixels) of the subtile

height – Height (in pixels) of the subtile
channelList – Array of channel indices.

Throws: TileFormatException
if the specified bounding box is outside of the parent tile.

Overrides:
createSubtile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile(int w,
                                 int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Parameters that are guaranteed to remain the same are:

- number of channels
- pixel stride

Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
        may not be equal to:
gc.createCompatibleTile(w2, h2)
```

Overrides:
createCompatibleTile in class Tile

o **createCompatibleTile**

```
public Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays. If the tile is a subtile, this will call createCompatibleTile(width, height).

Overrides:
createCompatibleTile in class Tile

o **toString**

```
public String toString()
```

Overrides:
toString in class Object

Class java.awt.image.ShortLookupTable

```
java.lang.Object
   |
   +----java.awt.image.LookupTable
           |
           +----java.awt.image.ShortLookupTable
```

public class **ShortLookupTable**
extends LookupTable

This class defines a lookup table object. The lookup table contains short data for one or more tile channels or image components (for example, separate arrays for R, G, and B), and it contains an offset which will be subtracted from the input value before indexing the array. This allows an array smaller than the native data size to be constructed for a constrained input. If there is only one array in the lookup table, it will be applied to all channels.

See Also:
ByteLookupTable, LookupOp

Constructor Index

o **ShortLookupTable**(int, short[])
Creates a LookupTable object from an array of short ints representing a lookup table for each tile channel.
o **ShortLookupTable**(int, short[][])
Creates a LookupTable object from an array of short int arrays representing a lookup table for each tile channel.

Method Index

o **getDataStorage**()
Returns the lookup table data.
o **lookupPixel**(int[], int[])
Returns an array of components of a pixel, translated with the lookup table.
o **lookupPixel**(short[], short[])
Returns an array of components of a pixel, translated with the lookup table.

Constructors

o **ShortLookupTable**

```
public ShortLookupTable(int offset,
                        short data[][])
```

Creates a LookupTable object from an array of short int arrays representing a lookup table for each tile channel. The offset will be subtracted from the input value before indexing into the arrays. The number of components is the length of the data argument. The data array for each component is stored as a reference.

o **ShortLookupTable**

```
public ShortLookupTable(int offset,
                        short data[])
```

Creates a LookupTable object from an array of short ints representing a lookup table for each tile channel. The offset will be subtracted from the input value before indexing into the array. The number of components is one. The data is copied to internal tables.

Methods

o **getDataStorage**

```
public short[][] getDataStorage()
```

Returns the lookup table data.

Returns:
LookupTable data array.

o **lookupPixel**

```
public int[] lookupPixel(int src[],
                         int dst[])
```

Returns an array of components of a pixel, translated with the lookup table. The source and destination can be equal. If dst is null, a new array will be allocated. The dst array is returned.

Returns:
an int array of components.
Overrides:
lookupPixel in class LookupTable

o **lookupPixel**

```
public short[] lookupPixel(short src[],
                           short dst[])
```

Returns an array of components of a pixel, translated with the lookup table. The source and destination can be equal. If dst is null, a new array will be allocated. The dst array is returned.

Returns:
a byte array of components.

Class java.awt.image.ThresholdOp

```
java.lang.Object
   |
   +----java.awt.image.ThresholdOp
```

public class **ThresholdOp**
extends Object
implements BufferedImageOp, TileOp

This class performs thresholding on the source image by mapping the value of each image component (for BufferedImages) or channel element (for Tiles) that falls between a low and a high value, to a constant.

For Tiles, the number of sets of threshold constants may be one, in which case the same constants are applied to all channels, or it must equal the number of Source tile channels.

For BufferedImages, the number of sets of threshold constants may be one, in which case the same constants are applied to all color and alpha components, or it must equal the number of Source color components, in which case no thresholding of the alpha component (if present) is performed, or it must equal the number of Source color components plus alpha components, in which case all color and alpha components are thresholded.

The pseudo code for the thresholding operation is as follows:

```
for each pixel from Source object {
    for each channel/component of the pixel {
        if (useLow) {
            if (srcElement <= thresholdValue) {
                destElement = lowValue;
            }
        }
        if (useHigh) {
            if (srcElement > thresholdValue) {
                destElement = highValue;
            }
        }
    }
}
```

Note that in−place operation is allowed (i.e. the source and destination can be the same object).

Constructor Index

o **ThresholdOp**(float, float, float)
Constructs a new ThresholdOp with the desired mapping.
o **ThresholdOp**(float, float[], float[])
Constructs a new ThresholdOp with the desired per channel/component mappings.

Method Index

o **createCompatibleDestImage**(BufferedImage, ColorModel)
Creates an empty destination image with the correct size and number of channels.
o **createCompatibleDestTile**(Tile)
Creates an empty destination tile with the correct size and number of channels.
o **filter**(BufferedImage, BufferedImage)
Performs thresholding on the BufferedImage.
o **filter**(Tile, Tile)
Performs thresholding on the Tile.
o **getDestBounds**(BufferedImage)
Returns the bounding box of the thresholded destination.
o **getDestBounds**(Tile)
Returns the bounding box of the thresholded destination.
o **getDestPoint**(Point2D, Point2D)
Returns the location of the destination point given a point in the source image.
o **getHighs**()
Returns the high value array.
o **getLows**()
Returns the low value array.

Constructors

o **ThresholdOp**

```
public ThresholdOp(float threshold,
                   float low[],
                   float high[])
```

Constructs a new ThresholdOp with the desired per channel/component mappings. The number of sets of mapping constants is subject to the restrictions given in the class comments above.

o **ThresholdOp**

```
public ThresholdOp(float threshold,
                   float low,
                   float high)
```

Constructs a new ThresholdOp with the desired mapping. The same set of thresholding constants will be applied to all channels/components of the Tile/BufferedImage.

Methods

o **getLows**

```
public float[] getLows()
```

Returns the low value array. Can be null.

o **getHighs**

```
public float[] getHighs()
```

Returns the high value array.

o **filter**

```
public BufferedImage filter(BufferedImage src,
                            BufferedImage dest)
```

Performs thresholding on the BufferedImage. If the color model in the source image is not the same as that in the destination image, the pixels will be converted in the destination. If the destination image is null, a BufferedImage will be created with the source ColorModel. The number of sets of threshold constants in this object may be one, in which case the same constants are applied to all color and alpha components, or it must equal the number of Source color components, in which case no thresholding of the alpha component (if present) is performed, or it must equal the number of Source color components plus alpha components, in which case all color and alpha components are thresholded. Otherwise an IllegalArgumentException will be thrown.

o **filter**

```
public Tile filter(Tile src,
                   Tile dest)
```

Performs thresholding on the Tile. If the destination Tile is null, a new Tile will be created. The source and destination must have the same number of channels. The number of sets of threshold constants in this object may be one, in which case the same constants are applied to all channels, or it must equal the number of Source tile channels. An IllegalArgumentException will be thrown if the number of channels in the source does not match the destination, or if the above restrictions on the number of sets of threshold constants are not met.

o **getDestBounds**

```
public Rectangle2D getDestBounds(BufferedImage src)
```

Returns the bounding box of the thresholded destination. Since this is not a geometric operation, the bounding box does not change. An IllegalArgumentException will be thrown if the number of sets of threshold constants does not meet the restrictions stated in the class comments above.

o **getDestBounds**

```
public Rectangle2D getDestBounds(Tile src)
```

Returns the bounding box of the thresholded destination. Since this is not a geometric operation, the bounding box does not change. An IllegalArgumentException will be thrown if the number of sets of threshold constants does not meet the restrictions stated in the class comments above.

o **createCompatibleDestImage**

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
                                               ColorModel destCM)
```

Creates an empty destination image with the correct size and number of channels. An IllegalArgumentException will be thrown if the number of sets of threshold constants does not meet the restrictions stated in the class comments above.

Parameters:

src – Source image for the filter operation.
destCM – ColorModel of the destination. If null, the ColorModel of the source will be used.

o **createCompatibleDestTile**

```
public Tile createCompatibleDestTile(Tile src)
```

Creates an empty destination tile with the correct size and number of channels. An IllegalArgumentException will be thrown if the number of sets of threshold constants does not meet the restrictions stated in the class comments above.

o **getDestPoint**

```
public Point2D getDestPoint(Point2D srcPt,
                            Point2D dstPt)
```

Returns the location of the destination point given a point in the source image. If dstPt is non–null, it will be used to hold the return value. Since this is not a geometric operation, the srcPt will equal the dstPt.

Class java.awt.image.Tile

Class java.awt.image.Tile

```
java.lang.Object
   |
   +----java.awt.image.Tile
```

---

public abstract class **Tile**
extends Object
super class of:
    ByteBandedTile, ByteComponentTile, BytePackedTile, IntegerComponentTile, ShortBandedTile, ShortComponentTile

This class defines a tile, which is a collection of channels in an image. A channel is a collection of data and layout parameters for one band of an image. There is no explicit channel object. The tile also holds subtiling information, such as the width, height, and data offsets of the subregion. All tiles have a base tile, which is the ancestor of a subtile whose base tile is itself. The data offsets are the same for all channels in a tile and are absolute pixel offsets into the channel data arrays (i.e. the offsets in a subtile are not relative to the offsets in the parent of that subtile).

Tile objects can be classified into two major categories: ComponentTiles, BandedTiles and PackedTiles. Both ComponentTiles and BandedTiles have only one channel element per data array element in each channel. ComponentTiles have only one data array for all channels whereas BandedTiles have n arrays for n channels. When accessing regions of pixels, the methods on ComponentTiles will tend to access the data channel−interleaved while methods on BandedTiles will access the data channel−sequential. PackedTiles have only one channel but more than one channel element may be packed into a data array element.

Glossary:

channel
: a collection of data and data layout parameters for one band of an image

tile
: a collection of channels in an image

data array
: a collection of Java primitive types (byte, short, int).

data array element
: one primitive type datum out of the data array for a channel

channel element
: Datum at an x,y location for one channel. For ComponentTile or BandedTile classes, there is only one channel element in one data array element. For PackedTile classes, there may be

multiple channel elements packed in one data array element.

pixel
: a collection of channel elements for a given x,y location

See Also:
: IntegerComponentTile, ShortComponentTile, ByteComponentTile, ShortBandedTile, ByteBandedTile, BytePackedTile

---

Constructor Index

o **Tile**(int, int, int, int, int)
: Constructs a base tile given the channel array and tile dimensions.

Method Index

o **createCompatibleTile**()
: Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.

o **createCompatibleTile**(int, int)
: Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays.

o **createSubtile**(int, int, int, int)
: Creates a subtile given a region of the tile.

o **createSubtile**(int, int, int, int, int[])
: Creates a subtile given a region of the tile.

o **getBaseTile**()
: Returns the base tile, which is the topmost tile from which any subtiles are created..

o **getData**(int, int, int)
: Returns the channel element for the specified channel at the specified location.

o **getData**(int, int, int, int, int, int[])
: Returns an integer array of pixels from the specified rectangular region.

o **getData**(int, int, int, int, int[])
: Returns an integer array of pixels from the specified rectangular region.

o **getData**(int, int, int[])
: Returns the channel elements for all channels at the specified location.

o **getHeight**()
: Returns the height (in pixels) of this tile.

o **getNumChannels**()
: Returns the number of channels.

o **getWidth**()
: Returns the width (in pixels) of this tile.

o **getXOffset**()
: Returns the X coordinate of the offset into the channel data arrays.

o **getYOffset**()
: Returns the Y coordinate of the offset into the channel data arrays.

o **putData**(int, int, int, int)
Stores the channel element for the specified channel at the specified location.
o **putData**(int, int, int, int, int, int[])
Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int, int, int[])
Stores an integer array of pixels into the specified rectangular region.
o **putData**(int, int, int[])
Stores the channel elements for all channels at the specified location.
o **putData**(int, int, Tile)
Stores the Tile data at the specified location.
o **setBaseTile**(Tile)
Set the base tile.

Constructors

o **Tile**

```
protected Tile(int x,
               int y,
               int width,
               int height,
               int numChannels)
```

Constructs a base tile given the channel array and tile dimensions.

Parameters:
x – X offset in the data array of each channel.
y – Y offset in the data array of each channel.
width – Width (in pixels) of the tile
height – Height (in pixels) of the tile
channels – Channel array.
Throws: TileFormatException
if the channel parameters are incompatible with the tile x offset, y offset, width and height.

Methods

o **getXOffset**

```
public int getXOffset()
```

Returns the X coordinate of the offset into the channel data arrays. Offsets for subtiles are not relative to the parent tile but are absolute offsets with respect to the base tile's data arrays.

o **getYOffset**

```
public int getYOffset()
```

Returns the Y coordinate of the offset into the channel data arrays. Offsets for subtiles are not relative to the parent tile but are absolute offsets with respect to the base tile's data arrays.

o **getWidth**

```
public int getWidth()
```

Returns the width (in pixels) of this tile.

o **getHeight**

```
public int getHeight()
```

Returns the height (in pixels) of this tile.

o **getNumChannels**

```
public int getNumChannels()
```

Returns the number of channels.

o **getBaseTile**

```
public Tile getBaseTile()
```

Returns the base tile, which is the topmost tile from which any subtiles are created.. (On creation of a tile, the base tile is set to itself.)

See Also:
createSubtile, createSubtile

o **setBaseTile**

```
protected void setBaseTile(Tile baseTile)
```

Set the base tile.

o **getData**

```
public abstract int getData(int x,
                            int y,
                            int channel)
```

Returns the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.

y – Y coordinate.
channel – Channel index.

o **getData**

```
public abstract int[] getData(int x,
                              int y,
                              int outData[])
```

Returns the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
outData – If non–null, channel elements for all channels at the specified location are returned in this array.
Returns:
Data array with channel elements for all channels.

o **getData**

```
public abstract int[] getData(int x,
                              int y,
                              int w,
                              int h,
                              int channel,
                              int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
// To find the channel element at location (x2, y2)
int chanElement = channelData[((y2-y)*w + (x2-x))];
```

o **getData**

```
public abstract int[] getData(int x,
                              int y,
                              int w,
                              int h,
                              int outData[])
```

Returns an integer array of pixels from the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

```
int[] channelData = tile.getData(x, y, w, h, null);
```

```
int numChannels = tile.getNumChannels();
int[] pixel = new int[numChannels];
// To find a pixel at location (x2, y2)
System.arraycopy(channelData, ((y2-y)*w + (x2-x))*numChannels,
                 pixel, 0, numChannels);
```

o **putData**

```
public abstract void putData(int x,
                             int y,
                             int channel,
                             int inData)
```

Stores the channel element for the specified channel at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
channel – Channel index.
inData – Channel element.

o **putData**

```
public abstract void putData(int x,
                             int y,
                             int inData[])
```

Stores the channel elements for all channels at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.
y – Y coordinate.
inData – Channel elements for all channels.

o **putData**

```
public void putData(int x,
                    int y,
                    Tile inTile)
```

Stores the Tile data at the specified location. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed.

Parameters:
x – X coordinate.

y – Y coordinate.
inTile – Tile of data to place at x,y location.

o **putData**

```
public abstract void putData(int x,
                             int y,
                             int w,
                             int h,
                             int channel,
                             int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **putData**

```
public abstract void putData(int x,
                             int y,
                             int w,
                             int h,
                             int inData[])
```

Stores an integer array of pixels into the specified rectangular region. There will be no explicit bounds checking on the parameters. An ArrayOutOfBounds exception will be thrown at runtime if data outside of the array is accessed. The channel elements in the data array are assumed to be packed. That is, a channel element for the nth channel at location (x2, y2) would be found at:

```
inData[((y2-y)*w + (x2-x))*numChannels + n]
```

o **createSubtile**

```
public abstract Tile createSubtile(int x,
                                   int y,
                                   int width,
                                   int height)
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper–left corner of this tile to the upper–left corner of the subtile. Note that the subtile will reference the same data arrays as the parent tile, but using different offsets.

Parameters:
x – X offset.

y – Y offset.
width – Width of the subtile.
height – Height of the subtile.

Throws: TileFormatException

if the specified bounding box is outside of the parent tile.

o **createSubtile**

```
public abstract Tile createSubtile(int x,
                                   int y,
                                   int w,
                                   int h,
                                   int channelList[])
```

Creates a subtile given a region of the tile. The x and y coordinates specify the horizontal and vertical offsets from the upper–left corner of this tile to the upper–left corner of the subtile. A subset of the parent tile's channels may be specified. Note that the subtile will reference the same data arrays as the parent tile, but using different offsets.

Parameters:

x – X offset.
y – Y offset.
width – Width of the data.
height – Height of the data.
channelList – Array of channel indices.

Throws: TileFormatException

if the specified bounding box is outside of the parent tile.

o **createCompatibleTile**

```
public abstract Tile createCompatibleTile(int w,
                                          int h)
```

Creates a tile with the same channel layout but using a different width and height, and with new zeroed data arrays. Note that given a GraphicsConfiguration object, gc,

```
gc.createCompatibleTile(w1, h1).createCompatibleTile(w2, h2)
        may not be equal to:
gc.createCompatibleTile(w2, h2)
```

o **createCompatibleTile**

```
public abstract Tile createCompatibleTile()
```

Creates a tile with the same channel layout and the same width and height, and with new zeroed data arrays.

Class java.awt.image.TileFormatException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----java.lang.RuntimeException
                           |
                           +----java.awt.image.TileFormatException
```

public class **TileFormatException**
extends RuntimeException

This exception is thrown if there is invalid layout information in the Tile.

Constructor Index

o **TileFormatException**(String)

Constructors

o **TileFormatException**

```
public TileFormatException(String s)
```