

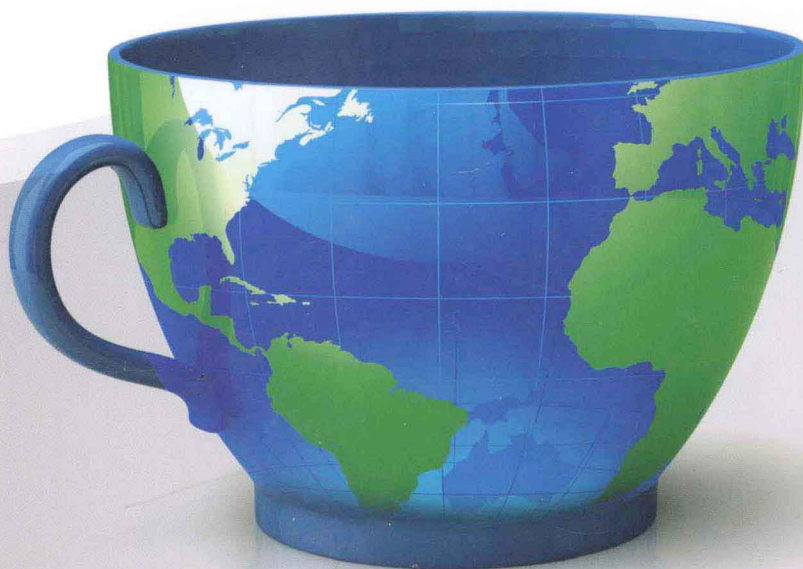
PEARSON

JAVATM 核心技术

Core Java Volume II — Advanced Features, Ninth Edition

卷 II：高级特性 下

(第9版·英文版)



[美] Cay S. Horstmann Gary Cornell 著

人民邮电出版社
POSTS & TELECOM PRESS

第2版 第2次印刷

JAVA 核心技术

Core Java Volume II—Advanced Features, Ninth Edition

卷 II：高级特性

（第9版）



© 2011 Pearson Education, Inc. All rights reserved.

机械工业出版社

JAVA 核心技术

Core Java Volume II — Advanced Features, Ninth Edition

卷 II：高级特性 下

（第9版·英文版）

[美] Cay S. Horstmann Gary Cornell 著

人 民 邮 电 出 版 社
北 京

Contents

Chapter 7: Advanced AWT	549
7.1 The Rendering Pipeline	550
7.2 Shapes	553
7.2.1 Using the Shape Classes	555
7.3 Areas	570
7.4 Strokes	572
7.5 Paint	581
7.6 Coordinate Transformations	583
7.7 Clipping	589
7.8 Transparency and Composition	592
7.9 Rendering Hints	601
7.10 Readers and Writers for Images	608
7.10.1 Obtaining Readers and Writers for Image File Types	608
7.10.2 Reading and Writing Files with Multiple Images	610
7.11 Image Manipulation	619
7.11.1 Constructing Raster Images	619
7.11.2 Filtering Images	626
7.12 Printing	636
7.12.1 Graphics Printing	637
7.12.2 Multiple-Page Printing	647
7.12.3 Print Preview	649
7.12.4 Print Services	659
7.12.5 Stream Print Services	664
7.12.6 Printing Attributes	664
7.13 The Clipboard	672
7.13.1 Classes and Interfaces for Data Transfer	674
7.13.2 Transferring Text	674
7.13.3 The Transferable Interface and Data Flavors	678
7.13.4 Building an Image Transferable	680
7.13.5 Transferring Java Objects via the System Clipboard	685
7.13.6 Using a Local Clipboard to Transfer Object References	689

7.14	Drag and Drop	689
7.14.1	Data Transfer Support in Swing	691
7.14.2	Drag Sources	696
7.14.3	Drop Targets	699
7.15	Platform Integration	707
7.15.1	Splash Screens	708
7.15.2	Launching Desktop Applications	713
7.15.3	The System Tray	719
Chapter 8: JavaBeans Components		725
8.1	Why Beans?	726
8.2	The Bean-Writing Process	728
8.3	Using Beans to Build an Application	731
8.3.1	Packaging Beans in JAR Files	731
8.3.2	Composing Beans in a Builder Environment	733
8.4	Naming Patterns for Bean Properties and Events	740
8.5	Bean Property Types	743
8.5.1	Simple Properties	744
8.5.2	Indexed Properties	744
8.5.3	Bound Properties	745
8.5.4	Constrained Properties	746
8.6	BeanInfo Classes	754
8.7	Property Editors	758
8.7.1	Writing Property Editors	762
8.7.1.1	String-Based Property Editors	762
8.7.1.2	GUI-Based Property Editors	765
8.8	Customizers	770
8.8.1	Writing a Customizer Class	772
8.9	JavaBeans Persistence	779
8.9.1	Using JavaBeans Persistence for Arbitrary Data	784
8.9.1.1	Writing a Persistence Delegate to Construct an Object	784
8.9.1.2	Constructing an Object from Properties	786
8.9.1.3	Constructing an Object with a Factory Method	787
8.9.1.4	Postconstruction Work	787
8.9.1.5	Transient Properties	788

8.9.2	A Complete Example for JavaBeans Persistence	791
Chapter 9: Security	803	
9.1	Class Loaders	804
9.1.1	The Class Loader Hierarchy	806
9.1.2	Using Class Loaders as Namespaces	808
9.1.3	Writing Your Own Class Loader	808
9.2	Bytecode Verification	816
9.3	Security Managers and Permissions	821
9.3.1	Java Platform Security	822
9.3.2	Security Policy Files	826
9.3.3	Custom Permissions	834
9.3.4	Implementation of a Permission Class	835
9.4	User Authentication	842
9.4.1	JAAS Login Modules	849
9.5	Digital Signatures	858
9.5.1	Message Digests	859
9.5.2	Message Signing	862
9.5.3	Verifying a Signature	865
9.5.4	The Authentication Problem	868
9.5.5	Certificate Signing	870
9.5.6	Certificate Requests	872
9.6	Code Signing	873
9.6.1	JAR File Signing	873
9.6.2	Software Developer Certificates	878
9.7	Encryption	880
9.7.1	Symmetric Ciphers	881
9.7.2	Key Generation	882
9.7.3	Cipher Streams	887
9.7.4	Public Key Ciphers	888
Chapter 10: Scripting, Compiling, and Annotation Processing	893	
10.1	Scripting for the Java Platform	894
10.1.1	Getting a Scripting Engine	894
10.1.2	Script Evaluation and Bindings	895
10.1.3	Redirecting Input and Output	898

10.1.4	Calling Scripting Functions and Methods	899
10.1.5	Compiling a Script	901
10.1.6	An Example: Scripting GUI Events	901
10.2	The Compiler API	907
10.2.1	Compiling the Easy Way	907
10.2.2	Using Compilation Tasks	907
10.2.3	An Example: Dynamic Java Code Generation	913
10.3	Using Annotations	919
10.3.1	An Example: Annotating Event Handlers	920
10.4	Annotation Syntax	926
10.5	Standard Annotations	931
10.5.1	Annotations for Compilation	932
10.5.2	Annotations for Managing Resources	932
10.5.3	Meta-Annotations	933
10.6	Source-Level Annotation Processing	935
10.7	Bytecode Engineering	943
10.7.1	Modifying Bytecodes at Load Time	949
Chapter 11: Distributed Objects		953
11.1	The Roles of Client and Server	954
11.2	Remote Method Calls	957
11.2.1	Stubs and Parameter Marshalling	957
11.3	The RMI Programming Model	959
11.3.1	Interfaces and Implementations	959
11.3.2	The RMI Registry	961
11.3.3	Deploying the Program	965
11.3.4	Logging RMI Activity	968
11.4	Parameters and Return Values in Remote Methods	970
11.4.1	Transferring Remote Objects	971
11.4.2	Transferring Nonremote Objects	971
11.4.3	Dynamic Class Loading	974
11.4.4	Remote References with Multiple Interfaces	979
11.4.5	Remote Objects and the equals, hashCode, and clone Methods	980
11.5	Remote Object Activation	980

Chapter 12: Native Methods	989
12.1 Calling a C Function from a Java Program	990
12.2 Numeric Parameters and Return Values	997
12.2.1 Using printf for Formatting Numbers	997
12.3 String Parameters	999
12.4 Accessing Fields	1005
12.4.1 Accessing Instance Fields	1005
12.4.2 Accessing Static Fields	1009
12.5 Encoding Signatures	1010
12.6 Calling Java Methods	1012
12.6.1 Instance Methods	1012
12.6.2 Static Methods	1016
12.6.3 Constructors	1017
12.6.4 Alternative Method Invocations	1018
12.7 Accessing Array Elements	1019
12.8 Handling Errors	1023
12.9 Using the Invocation API	1028
12.10 A Complete Example: Accessing the Windows Registry	1034
12.10.1 Overview of the Windows Registry	1034
12.10.2 A Java Platform Interface for Accessing the Registry	1036
12.10.3 Implementation of Registry Access Functions as Native Methods	1036
<i>Index</i>	<i>1051</i>

Advanced AWT

In this chapter:

- The Rendering Pipeline, page 550
- Shapes, page 553
- Areas, page 570
- Strokes, page 572
- Paint, page 581
- Coordinate Transformations, page 583
- Clipping, page 589
- Transparency and Composition, page 592
- Rendering Hints, page 601
- Readers and Writers for Images, page 608
- Image Manipulation, page 619
- Printing, page 636
- The Clipboard, page 672
- Drag and Drop, page 689
- Platform Integration, page 707

You can use the methods of the `Graphics` class to create simple drawings. Those methods are sufficient for simple applets and applications, but they fall short when you create complex shapes or when you require complete control over the

appearance of the graphics. The Java 2D API is a more sophisticated class library that you can use to produce high-quality drawings. In this chapter, we will give you an overview of that API.

We'll then turn to the topic of printing and show how you can implement printing capabilities in your programs.

We will cover two techniques for transferring data between programs: the system clipboard and the drag-and-drop mechanism. You can use these techniques to transfer data between two Java applications or between a Java application and a native program. Finally, we cover techniques for making Java applications feel more like native applications, such as providing a splash screen and an icon in the system tray.

7.1 The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the `Graphics` class such as `drawRect` or `fillOval`. The Java 2D API supports many more options.

- You can easily produce a wide variety of *shapes*.
- You have control over the *stroke*—the pen that traces shape boundaries.
- You can *fill* shapes with solid colors, varying hues, and repeating patterns.
- You can use *transformations* to move, scale, rotate, or stretch shapes.
- You can *clip* shapes to restrict them to arbitrary areas.
- You can select *composition rules* to describe how to combine the pixels of a new shape with existing pixels.
- You can give *rendering hints* to make trade-offs between speed and drawing quality.

To draw a shape, you need to go through the following steps:

1. Obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 1.2, methods such as `paint` and `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    . . .
}
```

2. Use the `setRenderingHints` method to set *rendering hints*—trade-offs between speed and drawing quality.

```
RenderingHints hints = . . .;  
g2.setRenderingHints(hints);
```

3. Use the `setStroke` method to set the *stroke*. The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.

```
Stroke stroke = . . .;  
g2.setStroke(stroke);
```

4. Use the `setPaint` method to set the *paint*. The paint fills areas such as the stroke path or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

```
Paint paint = . . .;  
g2.setPaint(paint);
```

5. Use the `clip` method to set the *clipping region*.

```
Shape clip = . . .;  
g2.clip(clip);
```

6. Use the `transform` method to set a *transformation* from user space to device space. Use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

```
AffineTransform transform = . . .;  
g2.transform(transform);
```

7. Use the `setComposite` method to set a *composition rule* that describes how to combine the new pixels with the existing pixels.

```
Composite composite = . . .;  
g2.setComposite(composite);
```

8. Create a shape. The Java 2D API supplies many shape objects and methods to combine shapes.

```
Shape shape = . . .;
```

9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

```
g2.draw(shape);  
g2.fill(shape);
```

Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context; change the settings only if you want to deviate from the defaults.

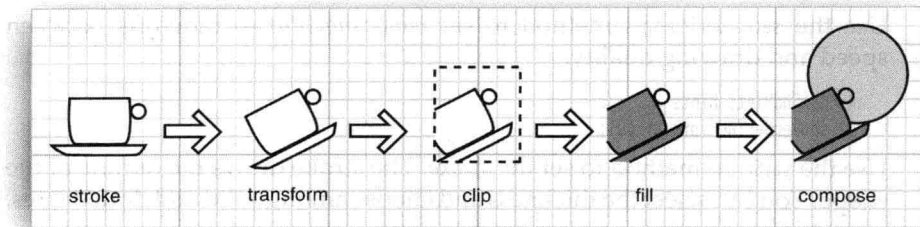


Figure 7.1 The rendering pipeline

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various set methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct Shape objects, no drawing takes place. A shape is only rendered when you call `draw` or `fill`. At that time, the new shape is computed in a *rendering pipeline* (see Figure 7.1).

In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.
2. The shape is transformed.
3. The shape is clipped. If there is no intersection between the shape and the clipping area, the process stops.
4. The remainder of the shape after clipping is filled.
5. The pixels of the filled shape are composed with the existing pixels. (In Figure 7.1, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

In the next section, you will see how to define shapes. Then, we will turn to the 2D graphics context settings.

java.awt.Graphics2D 1.2

- `void draw(Shape s)`
draws the outline of the given shape with the current paint.
- `void fill(Shape s)`
fills the interior of the given shape with the current paint.

7.2 Shapes

Here are some of the methods in the `Graphics` class to draw shapes:

```
drawLine  
drawRectangle  
drawRoundRect  
draw3DRect  
drawPolygon  
drawPolyline  
drawOval  
drawArc
```

There are also corresponding fill methods. These methods have been in the `Graphics` class ever since JDK 1.0. The Java 2D API uses a completely different, object-oriented approach. Instead of methods, there are classes:

```
Line2D  
Rectangle2D  
RoundRectangle2D  
Ellipse2D  
Arc2D  
QuadCurve2D  
CubicCurve2D  
GeneralPath
```

These classes all implement the `Shape` interface.

Finally, the `Point2D` class describes a point with an x and a y coordinate. Points are used to define shapes, but they aren't themselves shapes.

To draw a shape, first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class.

The `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D`, and `Arc2D` classes correspond to the `drawLine`, `drawRectangle`, `drawRoundRect`, `drawOval`, and `drawArc` methods. (The concept of a “3D rectangle” has died the death that it so richly deserved—there is no analog to the `draw3DRect` method.) The Java 2D API supplies two additional classes, quadratic and cubic curves, that we will discuss in this section. There is no `Polygon2D` class; instead, the `GeneralPath` class describes paths made up from lines, quadratic and cubic curves. You can use a `GeneralPath` to describe a polygon; we'll show you how later in this section.

The classes

```
Rectangle2D  
RoundRectangle2D
```

Ellipse2D
Arc2D

all inherit from a common superclass `RectangularShape`. Admittedly, ellipses and arcs are not rectangular, but they have a *bounding rectangle* (see Figure 7.2).

Each of the classes with a name ending in “2D” has two subclasses for specifying coordinates as float or double quantities. In Volume I, you already encountered `Rectangle2D.Float` and `Rectangle2D.Double`.

The same scheme is used for the other classes, such as `Arc2D.Float` and `Arc2D.Double`.

Internally, all graphics classes use float coordinates because float numbers use less storage space but have sufficient precision for geometric computations. However, the Java programming language makes it a bit more tedious to manipulate float numbers. For that reason, most methods of the graphics classes use double parameters and return values. Only when constructing a 2D object must you choose between the constructors with float and double coordinates. For example,

```
Rectangle2D floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);  
Rectangle2D doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

The `Xxx2D.Float` and `Xxx2D.Double` classes are subclasses of the `Xxx2D` classes. After object construction, essentially no benefit accrues from remembering the subclass, and you can just store the constructed object in a superclass variable as in the code example above.

As you can see from the curious names, the `Xxx2D.Float` and `Xxx2D.Double` classes are also inner classes of the `Xxx2D` classes. That is just a minor syntactical convenience to avoid inflation of outer class names.

Figure 7.3 shows the relationships between the shape classes. However, the `Double` and `Float` subclasses are omitted. Legacy classes from the pre-2D library are marked with a gray fill.

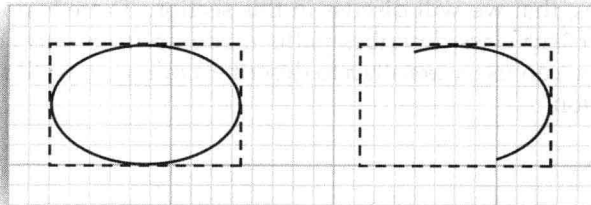


Figure 7.2 The bounding rectangle of an ellipse and an arc

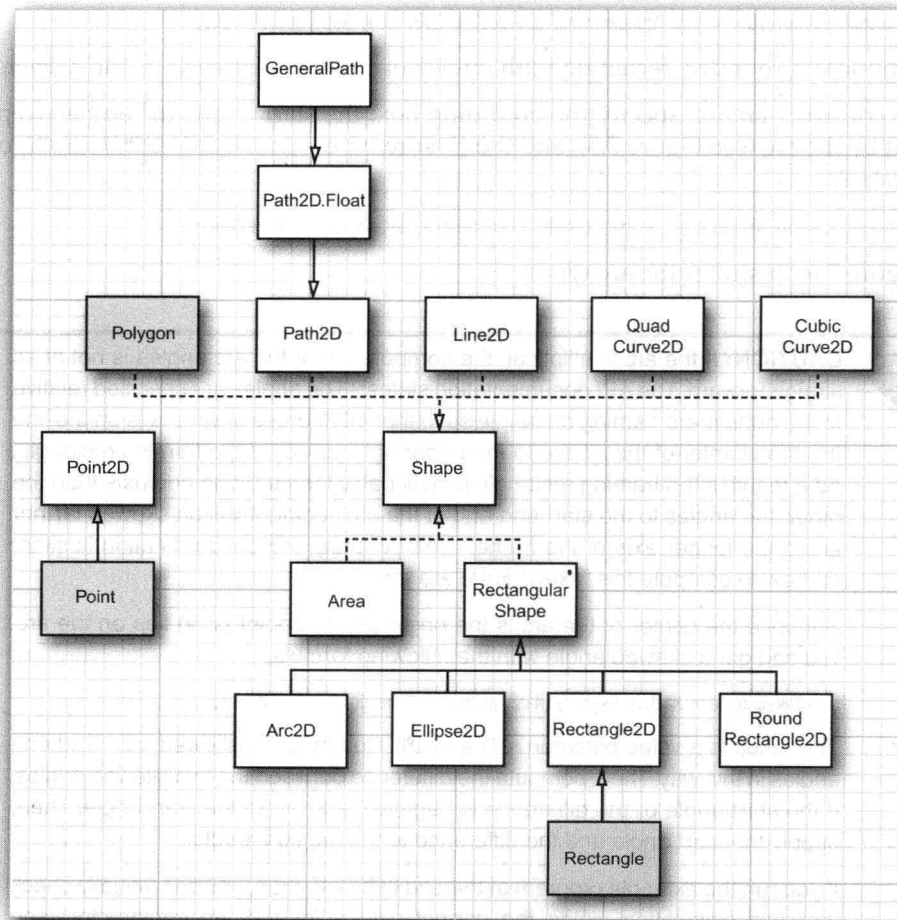


Figure 7.3 Relationships between the shape classes

7.2.1 Using the Shape Classes

You already saw how to use the `Rectangle2D`, `Ellipse2D`, and `Line2D` classes in Volume I, Chapter 7. In this section, you will learn how to work with the remaining 2D shapes.

For the `RoundRectangle2D` shape, specify the top left corner, width, height, and the x and y dimensions of the corner area that should be rounded (see Figure 7.4). For example, the call

```
RoundRectangle2D r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

produces a rounded rectangle with circles of radius 20 at each of the corners.

To construct an arc, specify the bounding box, the start angle, the angle swept out by the arc (see Figure 7.5), and the closure type—one of `Arc2D.OPEN`, `Arc2D.PIE`, or `Arc2D.CHORD`.

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureType);
```

Figure 7.6 illustrates the arc types.



CAUTION: If the arc is elliptical, the computation of the arc angles is not at all straightforward. The API documentation states: “The angles are specified relative to the nonsquare framing rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the framing rectangle. As a result, if the framing rectangle is noticeably longer along 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 frame.” Unfortunately, the documentation is silent on how to compute this “skew.” Here are the details:

Suppose the center of the arc is the origin and the point (x, y) lies on the arc. You can get a skewed angle with the following formula:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

The result is a value between -180 and 180 . Compute the skewed start and end angles in this way. Then, compute the difference between the two skewed angles. If the start angle or the difference is negative, add 360 to the start angle. Then, supply the start angle and the difference to the arc constructor.

If you run the example program at the end of this section, you can visually check that this calculation yields the correct values for the arc constructor (see Figure 7.9 on p. 561).

The Java 2D API supports *quadratic* and *cubic* curves. In this chapter, we do not get into the mathematics of these curves. We suggest you get a feel for how the curves look by running the program in Listing 7.1. As you can see in Figures 7.7 and 7.8, quadratic and cubic curves are specified by two *end points* and one or two *control points*. Moving the control points changes the shape of the curves.

To construct quadratic and cubic curves, give the coordinates of the end points and the control points. For example,

```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY, controlX, controlY, endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(startX, startY, control1X, control1Y,
    control2X, control2Y, endX, endY);
```

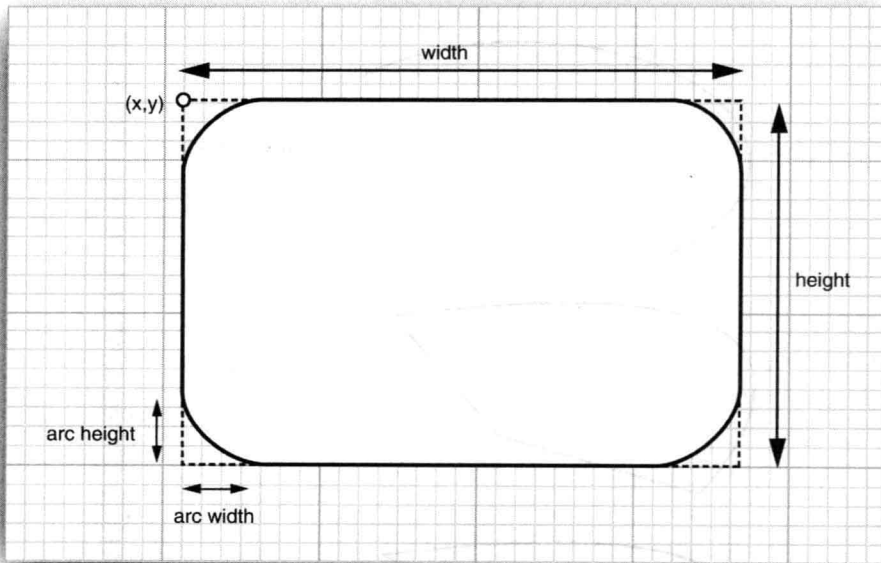


Figure 7.4 Constructing a `RoundRectangle2D`

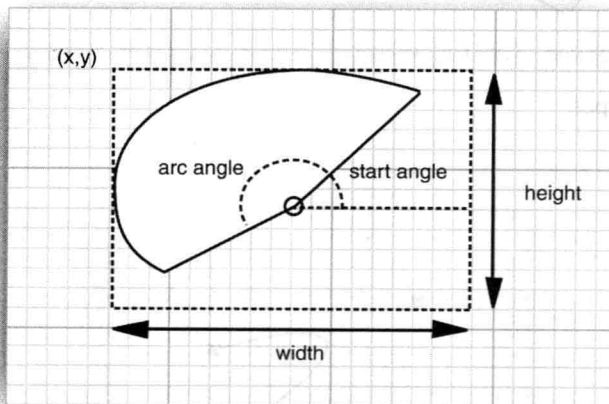


Figure 7.5 Constructing an elliptical arc