JClass PageLayout™

Programmer’s Guide

Version 6.3
for Java 2 (JDK 1.3.1 and higher)

The Best Way to Add Printing and Web Publishing to Your Java Applications

QUEST SOFTWARE™

8001 Irvine Center Drive
Irvine, CA 92618
949-754-8000
www.quest.com

April 2004

RefNo: PRGDE-JCPL/630-04/2004

Warranty

The information contained in this document is subject to change without notice. Quest Software makes no warranty of any kind with respect to this information. QUEST SOFTWARE SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTY OF THE MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Quest Software shall not be liable for any direct, indirect, incidental, consequential, or other damage alleged in connection with the furnishing or use of this information.

Trademarks

World Headquarters
8001 Irvine Center Drive
Irvine, CA 92618
www.quest.com
e-mail: info@quest.com
U.S. and Canada: 949.754.8000

Please refer to our Web site for regional and international office information.

This product includes software developed by the Apache Software Foundation http://www.apache.org/.

Table of Contents

Part I: Using JClass PageLayout

Part II:

Reference Appendices

Preface

Introducing JClass PageLayout

JClass PageLayout offers Java developers a set of methods and procedures for adding paginated, formatted, flowed-text, and image output to Java applications.
JClass PageLayout supports Western European languages that use ISO Latin-1 fonts.

You can use JClass PageLayout to build applications that create sophisticated print output involving columns, frames, multiple fonts and layout styles, pre-defined headers and footers, and more.

You can freely distribute Java applets and applications containing JClass components according to the terms of the License Agreement.

Feature Overview

JClass PageLayout creates print output from your Java application. It can send the output to your system printer, display it in a print preview window, or format it into a PCL, PostScript, or PDF file.

JClass PageLayout may be used in conjunction with Swing components and the rest of Quest's JClass product line. This means that you can build print output into any application created with Swing and JClass. For example, if your application generates a sales chart, you could print it on your system printer or format it as a PDF file.

Assumptions

This manual assumes that you have some experience with the Java programming language. You should have a basic understanding of object-oriented programming and Java programming concepts such as classes, methods, and packages before proceeding with this manual. See Related Documents later in this section of the manual for additional sources of Java-related information.

Typographical Conventions in this Manual

`Typewriter Font`	■ Java language source code and examples of file contents. ■ JClass PageLayout and Java classes, objects, methods, properties, constants, and events. ■ HTML documents, tags, and attributes. ■ Commands that you enter on the screen.
Italic Text	■ Pathnames, filenames, URLs, programs, and method parameters. ■ New terms as they are introduced, and to emphasize important words. ■ Figure and table titles. ■ The names of other documents referenced in this manual, such as *Java in a Nutshell*.
Bold	■ Keyboard key names and menu references.

Overview of the Manual

Part I – Using JClass PageLayout – describes programming with JClass PageLayout.

Chapter 1, JClass PageLayout Basics, is an introduction to writing JClass PageLayout programs, including a sample *Hello, World* program, the basic steps for writing a program, a description of the objects used in JClass PageLayout, and information on rendering and printing text.

Chapter 2, Creating a Document, guides you through the mechanics of outputting a print document from JClass PageLayout, including building page templates, instantiating printer and document objects, and controlling text flow.

Chapter 3, Formatting Text, shows you how to program text layout by creating text styles, changing fonts, modifying paragraphs, and inserting tab stops.

Chapter 4, Creating Tables, provides instructions for inserting tables, flowing data into tables, customizing table appearance by applying borders and background colors, wrapping tables, and converting other types of Java tables into the table type used by JClass PageLayout.

Chapter 5, Adding Formulas to JClass PageLayout, provides information on using the mathematical operations in *com.klg.jclass.util.formulae*. They extend the mathematical capabilities of standard Java to include objects that function as variables that may be scalars, vectors, matrices, or lists of expressions. If the targets referred to change dynamically, the mathematical operation updates the result upon re-evaluation.

Chapter 6, Refining a Document, helps you program more sophisticated documents by adding headers and footers, splitting a page into multiple columns, inserting page numbers, drawing geometric shapes, and importing image files.

Chapter 7, Printing Options, describes your print output options, including printing to the system printer, printing to an HTML, PostScript, PDF, or PCL file, printing to another screen in your application, and opening the print output in print preview mode.

Part II – Reference Appendices – contains detailed technical reference information.

Appendix A, JClass PageLayout Design Elements, summarizes the design elements in JClass PageLayout.

Appendix B, JClass PageLayout Commonly Used Methods, outlines the methods used most often in JClass PageLayoutAPI Reference

The API reference documentation (Javadoc) is installed automatically when you install JClass PageLayout and is found in the *JCLASS_HOME/docs/api/* directory.

Licensing

In order to use JClass PageLayout, you need a valid license. Complete details about licensing are outlined in the *JClass DesktopViews Installation Guide*, which is automatically installed when you install JClass PageLayout.

Related Documents

The following is a sample of useful references to Java and XML programming:

- "*Java Platform Documentation*" at *http://java.sun.com/docs/index.html* and the "*Java Tutorial*" at *http://java.sun.com/docs/books/tutorial/index.html* from Sun Microsystems.
- For an introduction to creating enhanced user interfaces, see "*Creating a GUI with JFC/Swing*" at *http://java.sun.com/docs/books/tutorial/uiswing/index.html*.
- *Java in a Nutshell, 2nd Edition* from O'Reilly & Associates Inc. See the O'Reilly Java Resource Center at *http://java.oreilly.com*.
- *http://www.w3.org/XML/* – another W3C site; contains exhaustive information on standards. Of particular note are the XML schema 1 (structures) and XML schema 2 (datatypes) working drafts. They make up an extension that specifies how to constrain XML documents to particular schema. This is important if you want to represent database data or object-oriented data as XML.
- *http://www.javasoft.com/xml/tutorial_intro.html* – Sun's XML site.

These documents are not required to develop applications using JClass PageLayout, but they can provide useful background information on various aspects of the Java programming language.

About Quest

Quest Software, Inc. (NASDAQ: QSFT) is a leading provider of application management solutions. Quest provides customers with Application Confidence^{sm} by delivering reliable software products to develop, deploy, manage and maintain enterprise applications without expensive downtime or business interruption. Targeting high availability, monitoring, database management and Microsoft infrastructure management, Quest products increase the performance and uptime of business-critical applications and enable IT professionals to achieve more with fewer resources. Headquartered in Irvine, Calif., Quest Software has offices around the globe and more than 18,000 global customers, including 75% of the Fortune 500. For more information on Quest Software, visit *www.quest.com*.

Contacting Quest Software

E-mail	*sales@quest.com*
Address	Quest Software, Inc. World Headquarters 8001 Irvine Center Drive Irvine, CA 92618 USA
Web site	*www.quest.com*
Phone	949.754.8000 (United States and Canada)

Please refer to our Web site for regional and international office information.

Customer Support

Quest Software's world-class support team is dedicated to ensuring successful product installation and use for all Quest Software solutions.

SupportLink	*www.quest.com/support*
E-mail	*support@quest.com*

You can use SupportLink to do the following:

- Create, update, or view support requests
- Search the knowledge base, a searchable collection of information including program samples and problem/resolution documents

- Access FAQs
- Download patches
- Access product documentation, API reference, and demos and examples

Please note that many of the initial questions you may have will concern basic installation or configuration issues. Consult this product's *readme* file and the *JClass DesktopViews Installation Guide* (available in HTML and PDF formats) for help with these types of problems.

To Contact JClass Support

Any request for support *must* include your JClass product serial number. Supplying the following information will help us serve you better:

- Your name, email address, telephone number, company name, and country
- The product name, version and serial number
- The JDK (and IDE, if applicable) that you are using
- The type and version of the operating system you are using
- Your development environment and its version
- A full description of the problem, including any error messages and the steps required to duplicate it

JClass Direct Technical Support	
JClass Support Email	*support@quest.com*
Telephone	949-754-8000
Fax	949-754-8999
European Customers Contact Information	Telephone: +31 (0)20 510-6700 Fax: +31 (0)20 470-0326

Product Feedback and Announcements

We are interested in hearing about how you use JClass PageLayout, any problems you encounter, or any additional features you would find helpful. The majority of enhancements to JClass products are the result of customer requests.

Please send your comments to:
Quest Software
8001 Irvine Center Drive
Irvine, CA 92618

Telephone: 949-754-8000
Fax: 949-754-8999

Part I

Using JClass PageLayout

1

JClass PageLayout Basics

Overview of JClass PageLayout ■ *Basic Steps for Creating a JClass PageLayout Document*
A Simple JClass PageLayout Program ■ *JClass PageLayout Objects* ■ *Creating Flow* ■ *Printing*

1.1 Overview of JClass PageLayout

JClass PageLayout uses a flow-markup approach for creating multipage documents to be displayed or converted for printing. The major components of JClass PageLayout are:

- The `JCDocument` object, which stores document-level attributes and the list of completed pages (`JCPage` objects), and against which the `JCFlow` object is created.
- The flow mechanism, which is responsible for allocating content to pages and creating new pages as necessary.
- The `JCPrinter` object, which provides an abstraction across the set of supported output types, and whose subclasses provide the required instances of printer-specific implementations of Graphics2D for the conversion of draw actions to page-marking operands.

Other important top-level components are `JCTextStyle` (stores character- and paragraph-level attributes controlling the text appearance), `JCDrawStyle` (contains attributes of drawn geometric objects), and `JCPageTable` (describes general tables whose cells can contain any JClass PageLayout drawable including `JCPageTable`).

1.2 Basic Steps for Creating a JClass PageLayout Document

At a minimum, you need to perform the following steps to format and flow a JClass PageLayout document. For more information on any step, refer to the corresponding section of this guide.

Step...	See...
1. Select or create a page template.	Building Page Templates, in Chapter 2, and line 20, A Simple JClass PageLayout Program
2. Instantiate the printer.	Creating a Printer, in Chapter 2, and line 16, A Simple JClass PageLayout Program

3. Instantiate the document.	Creating a Document, in Chapter 2, and line 20, A Simple JClass PageLayout Program
4. Instantiate the flow.	Controlling Flow, in Chapter 2, and line 23, A Simple JClass PageLayout Program
5. Send the document to the specified printer.	Printing Options, in Chapter 7, and line 29, A Simple JClass PageLayout Program

1.3 A Simple JClass PageLayout Program

If you compile and run the following program, it generates a one-page PDF document containing the text “Hello, World.”

Note: For information on setting up your environment, please refer to the *JClass DesktopViews Installation Guide*.

```
 1 package examples.pagelayout;
 2
 3 import java.io.StringReader;
 4 import java.io.BufferedReader;
 5
 6 import com.klg.jclass.page.JCFlow;
 7 import com.klg.jclass.page.JCPrinter;
 8 import com.klg.jclass.page.JCDocument;
 9 import com.klg.jclass.page.adobe.pdf.JCPDFPrinter;
10
11 public class PdfPrinter {
12
13 public static void main(String[] args)
14
15 // Create a PDF printer, output to stdout
16 JCPrinter printer = new JCPDFPrinter(System.out);
17
18 // Create a document using the PDF printer for formatting,
19 // setting the page template to be a simple 8.5 x 11 Letter page
20 JCDocument document = new JCDocument(printer,
                                 JCDocument.BLANK_8p5X11);
21
22 // Instantiate a flow object on the document
23 JCFlow flow = new JCFlow(document);
24
25 // Print some text to the document
26 flow.print("Hello, World.");
27
28 // Print the document to the PDF printer
29 document.print();
30 }
31 }
```

Line 16 of the program instantiates the PDF printer to which the output of this file is sent. On line 20, we construct the document, at the same time specifying that the printer is the one created in line 16, and that the page template to use is the standard 8.5 x 11 that comes with JClass PageLayout. We then create the flow (line 23), render the text "Hello, World" to the flow (line 26), and send the document to the PDF printer (line 29).

1.4 JClass PageLayout Objects

JClass PageLayout uses a series of nested objects. `JCDocument` is the top-level object. `JCPage`, `JCFrame`, and `JCPageTable` objects are created, defined, and added to the `JCDocument` object. The `JCFlow` object controls how text and images are rendered into the `JCFrame` objects. When all text has been flowed into the `JCDocument` object, it is passed to a `JCPrinter` object which handles document output.

1.4.1 Class Overview

To use JClass PageLayout, you need to understand how its classes work together to create a document and output it. The following table provides a brief overview of the classes used in JClass PageLayout.

JClass PageLayout Class	Description
`JCDocument`	The `JCDocument` holds the `JCPage` and `JCFrame` objects into which you "flow" or "render" your text and images. Attributes specify how the flow should advance through the rendering process. Note that there can be only one `JCDocument` object per `JCFlow`; also, the `JCDocument` object must be the one passed to the `JCFlow` in the constructor.
`JCPage`	`JCPage` methods and attributes specify how an actual page should be laid out, what frames should exist on the page, and the frame order of the flow. `JCPage` objects are based on XML templates. Complex documents may need to define many different page templates. For example, separate templates would be necessary to describe a title page, a table of contents page, the first page in a chapter, and subsequent right and left body pages.
`JCFlow`	The `JCFlow` class provides methods and attributes that control how text and images are rendered to the document. There is only one flow in a document. To render text or images apart from the main flow, use `JCFrame` methods.
`JCFrame`	The frame is the basic unit of the flow. Frames contain all graphics (text, shapes, and images) rendered to the document. Complex documents may need to define many different `JCFrame` objects on a page. For example, most pages require a header frame, a body frame, and a footer frame.

JClass PageLayout Class	Description
JCTextStyle & JCDrawStyle	The style classes describe and control the appearance of text and figures in the document. A style object's attributes identify how the rendered text or images should appear, for example JCTextStyle.LINEMODE_UNDERLINE is used to underline text.
JCPageTable	The JCPageTable class provides methods and attributes for creating tables. Using the JCPageTable object's subclasses (Row, Column, and Cell), you can customize the table's rows, columns, and cells. Additional methods allow you to convert other types of Java tables, including JClass LiveTables, Swing JTables, and JDBC data.
JCTab	JCTab works with JCTextStyle and provides methods for modifying tab alignment, horizontal position, and appearance (tab leader).
JCPrinter	The JCPrinter class provides attributes and methods that control how the rendered document is printed on an output device: the number of pages to output, the number of copies, whether copies should be collated, and so on. JCPrinter also generates a list of Font Family and Font objects based on your printer drivers. A JCPrinter object needs to be created before the JCDocument object and passed to it. The JCDocument object uses the JCPrinter object's font information to properly render text.
formulas	You can add formulas to pages and to tables by employing the classes in com.klg.jclass.util.formulae. You build an expression that references page variables or table cells and performs one of a large number of mathematical operations on them, for example, summing a range of table cells. See Adding Formulas to JClass PageLayout, in Chapter 5, for details.

1.4.2 Objects on a Page

The following illustration demonstrates the output of our *Hello, World* program, found in Section 1.3, A Simple JClass PageLayout Program.

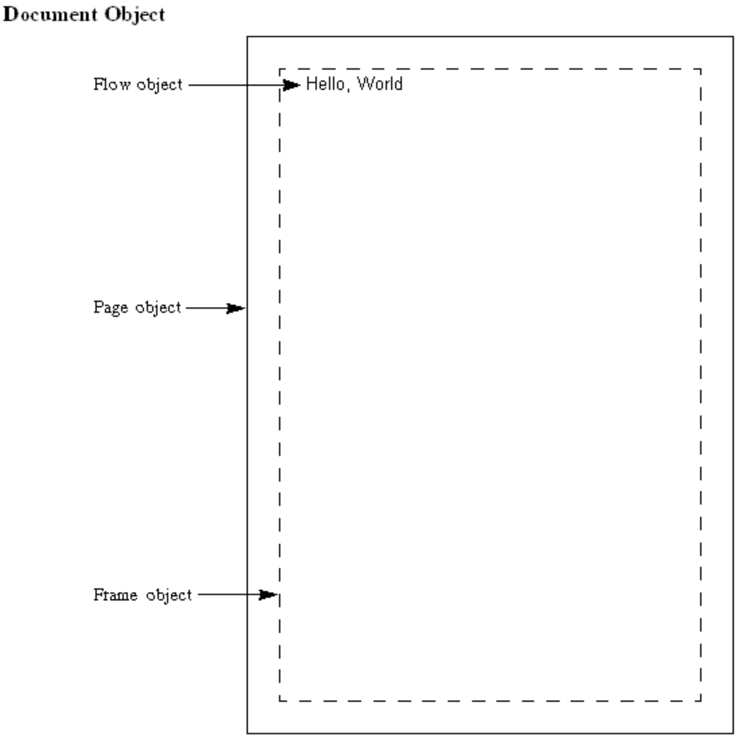


Figure 1 Example of how objects are used to build a page.

Figure 1 displays a `JCPage` object, which can only exist in a `JCDocument` object. The `JCPage` object contains the `JCFrame` object (`body`) that holds the text and the `JCFlow` object that renders the text.

1.5 Creating Flow

While the objects and template define where and how the text gets rendered, methods from the `JCFrame` and `JCFlow` classes control the flow of text from one frame to the next and from one page to the next.

When you want to control the flow of text throughout the document, use the `JCFlow` class. `JCFlow` manages layout from frame to frame and from page to page. When you want to control a frame layout or render text apart from the main flow of the document, use the `JCFrame` class. For more information on `JCFlow` and `JCFrame`, refer to Controlling Flow, in Chapter 2.

1.6 Printing

JClass PageLayout prints using `JCDocument.print()`. The print command can be as simple as:

```
doc.print();
```

JClass PageLayout can also print a page range. For example, to print pages 17 through 39, use:

```
doc.print(17, 39)
```

Before you can print the document, you must have set up a printer object, as described in Creating a Printer, in Chapter 2. The `JCHTMLPrinter`, `JCPCLPrinter`, `JCPDFPrinter`, and `JCPostScriptPrinter` objects print directly to a file, while the `JCAWTPrinter` object prints to your system printer.

When you use `JCAWTPrinter`, you may notice that the process of printing to your system printer is much slower than printing to a file. The Java 2 Printing API used by `JCAWTPrinter` causes large amounts of data to be sent to the system printer, resulting in the drastic slowdown. For more information on print output options and printing efficiency, refer to Printing Options, in Chapter 7.

Once a page is created, you can designate whether it will be held until the entire document is finished before it is printed. This is set via the `outputPolicy` property; please refer to OutputPolicy, in Chapter 7, for complete details.

As well, you can designate whether pages are to be discarded once printed. You can do this by setting the `flushPolicy` property; for information, please see FlushPolicy, in Chapter 7.

Printing Large Documents

When printing documents of any size, but especially for larger documents, better performance is achieved by wrapping the output stream in a `BufferedOutputStream`:

```
BufferedOutputStream bos = new BufferedOutputStream(os, 2048);
JCPrinter printer = new JC<type>Printer(bos);
```

This may be especially important in application server or web server environments where exceptions may be thrown if multiple writes are not buffered.

Printing Components

Components embedded into a JClass PageLayout document or encoded using the JClass PageLayout encoder types should not be double- buffered. If such components are double-buffered, then the components will draw to an image and the image will be drawn to the printer/file, thus resulting in a loss of resolution.

2

Creating a Document

In Basic Steps for Creating a JClass PageLayout Document, in Chapter 1, we outlined the five basic steps necessary to create the simplest document. In this chapter, we build on that foundation, giving you the specific information you need to make each of those steps.

2.1 Building Page Templates

In A Simple JClass PageLayout Program, in Chapter 1, you saw a JClass PageLayout program that outputs a single line of text to a printed page. Before you can flow content into a document, you need a page template that defines how the page is laid out. Page templates specify:

- the physical size of the page
- the location and size of the frames that will hold text and images
- the order text is to progress through those frames
- the next page to generate when the existing page and/or section is full

JClass PageLayout templates are written in the Extensible Markup Language (XML). The templates use a common Document Type Definition (DTD) that is built into the JClass PageLayout API. The DTD defines the tags and attributes used to specify the appearance of the page templates. It is located in *com/klg/jclass/xml-dtd/JCPageTemplate.dtd*.

For an introduction to XML, refer to: *http://www.javaworld.com/javaworld/jw-04-1999/jw-04-xml.html*. There are many guides that provide a more comprehensive examination of XML, such as *The XML Handbook*, by Charles Goldfarb and Paul Prescod.

JClass PageLayout provides default page templates for most standard page types. For the sake of simplicity, the *Hello, World* program uses a default page template, as follows:

```
doc = new JCDocument(printer, JCDocument.BLANK_8p5x11);
```

The following table lists and describes the default page templates available to you.

Default Template	Description
Blank_8p5x11	Creates a blank (no headers or footers) page of standard US Letter size.
Blank_8p5x14	Creates a blank (no headers or footers) page of standard US Legal size.
Blank_11x17	Creates a blank (no headers or footers) page of standard Tabloid size.
Blank_A3	Creates a blank (no headers or footers) page of standard ISO A3 size.
Blank_A4	Creates a blank (no headers or footers) page of standard ISO A4 size.
Blank_A5	Creates a blank (no headers or footers) page of standard ISO A5 size.

Unlike the introductory example in the previous chapter, most applications require more than one type of output page. If your application requires custom output pages, the template must provide a definition for each type of page.

2.1.1 A Sample Template

The following is an XML document you could use to define an 8.5 x 11 (US Letter) page template that, unlike the standard template Blank_8p5x11, includes headers and footers.

```
<?xml version="1.0"?>
<!DOCTYPE JCPAGETEMPLATE SYSTEM "JCPageTemplate.dtd">
<JCPAGETEMPLATE TITLE="8p5x11">
   <PAGE NAME="8p5x11" UNIT="inches">
       <LOCATION X="0" Y="0"/>
       <SIZE WIDTH="8.5" HEIGHT="11"/>
       <FRAME NAME="header" UNIT="inches" COLOR="grey">
           <LOCATION X="1" Y="0.25"/>
           <SIZE WIDTH="6.5" HEIGHT="0.75"/>
       </FRAME>
       <FRAME NAME="body" UNIT="inches">
           <LOCATION X="1" Y="1"/>
           <SIZE WIDTH="6.5" HEIGHT="9"/>
           <COLUMN COUNT="2"/>
       </FRAME>
       <FRAME NAME="footer" UNIT="inches" COLOR="pink">
           <LOCATION X="1" Y="10.25"/>
           <SIZE WIDTH="6.5" HEIGHT="0.75"/>
       </FRAME>
       <FLOWFRAME NAME="body"/>
       <FLOWPAGE NAME="8p5x11"/>
```

```
        <FLOWSECTION NAME="8p5x11"/>
    </PAGE>
</JCPAGETEMPLATE>
```

2.1.2 The Template DTD

The structure of JClass PageLayout templates is defined in an XML Document Type Definition (DTD) stored as a String in `JCPageTemplate`. We reprint it here to give you an idea of the hierarchy of elements in an XML template. For a complete description of the elements used in a JClass PageLayout template, refer to Template Elements and Attributes, in Chapter 2.

Please note that the order of the attributes and child elements of each element is fixed; the order shown in the example below must be used.

```
<!-- DTD for JClass PageLayout Templates -->
<!ELEMENT JCPAGETEMPLATE (PAGE+)>
<!ATTLIST JCPAGETEMPLATE TITLE CDATA #IMPLIED>

<!ELEMENT PAGE (LOCATION,SIZE,FRAME+,FLOWFRAME*,FLOWPAGE,FLOWSECTION)>
<!ATTLIST PAGE NAME CDATA #REQUIRED
   UNIT (inches|points|cm|cms|centimeters|centimetres) "inches"
   COLOR CDATA #IMPLIED
    ORIENTATION (automatic|portrait|landscape) "automatic"
   FIRST (True|true|TRUE|False|false|FALSE) "false">

<!ELEMENT FRAME (LOCATION,SIZE,BORDER?,COLUMN?,MARGIN?)>
<!ATTLIST FRAME NAME CDATA #REQUIRED
   UNIT (inches|points|cm|cms|centimeters|centimetres) "inches"
   COLOR CDATA #IMPLIED>

<!ELEMENT LOCATION EMPTY>
<!ATTLIST LOCATION X CDATA #REQUIRED
                   Y CDATA #REQUIRED>

<!ELEMENT SIZE EMPTY>
<!ATTLIST SIZE WIDTH CDATA #REQUIRED
               HEIGHT CDATA #REQUIRED>

<!ELEMENT BORDER EMPTY>
<!ATTLIST BORDER TYPE
(blank|broken|dashed|double|none|plain|regular|single) "blank"
   COLOR CDATA "black"
   THICKNESS CDATA #IMPLIED>

<!ELEMENT COLUMN EMPTY>
<!ATTLIST COLUMN COUNT CDATA #REQUIRED
   SPACING CDATA #IMPLIED>

<!ELEMENT MARGIN EMPTY>
<!ATTLIST MARGIN TOP CDATA #REQUIRED
                 RIGHT CDATA #REQUIRED
                 LEFT CDATA #REQUIRED
                 BOTTOM CDATA #REQUIRED>

<!ELEMENT FLOWFRAME EMPTY>
<!ATTLIST FLOWFRAME NAME CDATA #REQUIRED>

<!ELEMENT FLOWPAGE EMPTY>
<!ATTLIST FLOWPAGE NAME CDATA #REQUIRED>

<!ELEMENT FLOWSECTION EMPTY>
<!ATTLIST FLOWSECTION NAME CDATA #REQUIRED>
```

2.1.3 Template Elements and Attributes

The following table describes the elements and attributes used to define a page template.

Please note that the order of the attributes and child elements of each element is fixed; the order shown in the table below and in Section 2.1.2, The Template DTD, must be used.

Element	Attributes	Child Elements
<JCPAGETEMPLATE>	TITLE: An optional attribute that names this page template.	<PAGE>
<PAGE>	NAME: Required. The name of this page type, referenced by other page definitions using the <FLOWPAGE> tag. UNIT: The unit of measurement used to plot out this page. Choose from inches, points, cm, cms, centimetres, and centimeters. The default is inches. COLOR: Optional. Specifies a default background color for this page using hexadecimal notation or a color from com.klg.jclass.util.swing.JCSwingTypeConverter. ORIENTATION: Choose from automatic, portrait, and landscape. FIRST: A Boolean attribute that indicates whether or not this page template is used for the first page in the document. The default is false.	<LOCATION>, <SIZE>, <FRAME>+, <FLOWFRAME>*, <FLOWPAGE>, <FLOWSECTION>
<FRAME>	NAME: Required. The name of this frame type, referenced by other page definitions using the <FLOWFRAME> tag. UNIT: The unit of measurement used to plot out this frame. Choose from inches, points, cm, cms, centimetres, and centimeters. The default is inches. COLOR: Optional. Specifies a default background color for this frame using hexadecimal notation or a color from com.klg.jclass.util.swing.JCSwingTypeConverter.	<LOCATION>, <SIZE>, <BORDER>?, <COLUMN>?, <MARGIN>?
<LOCATION>	X: Required. Specifies the distance of the page or frame from the left-hand page edge. Y: Required. Specifies the distance of the page or frame from the top of the page.	None.

$^{+}$ Required and repeatable.
* Optional and repeatable.
$^{?}$ Optional and non-repeatable.

Element	Attributes	Child Elements
`<SIZE>`	`WIDTH`: Required. Specifies the width of the page or frame, measured in the units defined by the `<UNIT>` tag. `HEIGHT`: Required. Specifies the height of the page or frame, measured in the units defined by the `<UNIT>` tag.	None.
`<BORDER>`	`TYPE`: Specifies the style used to draw a frame border. Choose from `blank`, `broken`, `dashed`, `double`, `none`, `plain`, `regular`, or `single`. The default is `blank`. `COLOR`: Specifies the border color using either hexadecimal notation or a color from `java.awt.Color`. The default is `black`. `THICKNESS`: Optional. Specifies the border width in pixels. The default is `0.1`.	None.
`<COLUMN>`	`COUNT`: Required. Specifies the number of columns in the frame. `SPACING`: Optional. Specifies the amount of space left between columns, measured in the units defined by the `<UNIT>` tag.	None.
`<MARGIN>`	`TOP`: Required. Specifies the top margin. `RIGHT`: Required. Specifies the right margin. `LEFT`: Required. Specifies the left margin. `BOTTOM`: Required. Specifies the bottom margin.	None.
`<FLOWFRAME>`	`NAME`: Required. Specifies the name of a frame to be added to the sequence of frames to which the document will flow content.	None.
`<FLOWPAGE>`	`NAME`: Required. Specifies the name of the page to which the flow is to progress when a new page is begun.	None.
`<FLOWSECTION>`	`NAME`: Required. Specifies the name of the page to which the flow is to progress when a new section is begun.	None.

$^+$ Required and repeatable.
* Optional and repeatable.
$^?$ Optional and non-repeatable.

2.2 Applying Page Templates

After writing your own XML page template, you can apply it to your document by loading it either as an internal String or an external XML file. The following table describes the various `JCPageTemplate` methods you can use to apply templates.

Method	Explanation
`importTemplates(JCDocument doc, File xmlfile)`	Reads from `java.io.File` to import an XML template and apply it to the specified `JCDocument`.
`importTemplates(JCDocument doc, Reader reader)`	Reads from `java.io.Reader` to import an XML template and apply it to the specified `JCDocument`.
`importTemplates(JCDocument doc, InputSource input)`	Reads from `org.xml.sax.InputSource` to import an XML template and apply it to the specified `JCDocument.`
`loadTemplates(File xmlfile)`	Reads from `java.io.File` to load the XML template without applying it to a specific document.
`loadTemplates(Reader reader)`	Reads from `java.io.Reader` to load the XML template without applying it to a specific document.
`loadTemplates(InputSource input)`	Reads from `org.xml.sax.InputSource` to load the XML template without applying it to a specific document.

2.2.1 Loading External XML Files

Users of JClass PageLayout will need to include the JAR files *jaxp.jar* and *crimson.jar*[1] in their CLASSPATH. These files replace the now obsolete *parser.jar* and *xml.jar* from earlier releases. They are distributed in the *JCLASS_HOME/ lib* directory along with the JClass PageLayout JAR file.

The following example uses `java.io.File` to load the external XML file `8p5x11.xml` as a template.

```
JCDocument document = null;
try {
document = new JCDocument(printer, JCPageTemplate.loadTemplates
(new java.io.File("8p5x11.xml")));
}
catch (Exception e) {
System.err.println("Error loading template = " + e);
System.exit(1);
}
```

[1] You may substitute for *crimson.jar* any parser that is compliant with Sun's JAXP 1.1 specification. See Sun's JAXP documentation for more information:

2.2.2 Loading XML Strings

The following example uses java.io.StringReader to load an XML template that is defined as a String (template) earlier in the program.

```
JCDocument document = null;
try {
document = new JCDocument(printer, JCPageTemplate.loadTemplates
(new StringReader(template)));
}
catch (Exception e) {
System.err.println("Error loading template = " + e);
System.exit(1);
}
```

2.3 Creating a Printer

Now that the page templates are defined, the next step is to instantiate the Printer object that defines the type of print output produced.

```
// Open the output file
try {
outfile = new FileOutputStream("test.pdf");
}
catch (FileNotFoundException e) {
System.out.println("Could not open file");
return;
}

// Create a PDF printer
printer = new JCPDFPrinter(outfile);
```

This example creates a printer object which uses PostScript fonts, lays out the flow, and generates a PDF (Adobe Portable Document Format) file (test.pdf) when the document is printed.

Other printer types include JCHTMLPrinter, JCPostScriptPrinter and JCPCLPrinter, which create output in Adobe's PostScript and the Hewlett-Packard Printer Control Language (PCL), respectively. Using JCAWTPrinter, you can print to the system printer. JCAWTScreenPrinter is used for printing to a screen, which can be used with JCAWTPreviewer for print preview mode. For more information on printing, refer to Printing Options, in Chapter 7.

2.4 Creating a Document

The JCDocument object holds the JCPage and JCFrame objects into which text and images are flowed. In the *Hello, World* example, we defined the JCDocument as follows:

```
JCDocument document = new JCDocument(printer,
    JCDocument.BLANK_8p5X11);
```

When you instantiate a new JCDocument, you can specify the printer to which you want to send its output, along with the name of the standard template it is to use as shown in the code line above, or by using the JCPageTemplate.loadTemplates() method and passing a list of page templates. For information on defining a printer, refer to Creating a Printer, in Chapter 2. For information on standard templates, refer to Building Page Templates, in Chapter 2.

2.5 Controlling Flow

In cases where text is to flow from frame to frame, the page template specifies the basic flow of text within a document. Template Elements and Attributes, in Chapter 2, describes how to use FlowFrameList to direct the flow of text through frames, FlowPageTemplate to direct the flow of text to the next page, and FlowSectionTemplate to direct the flow of text after a section break.

At times, you may want to advance the flow before it reaches the end of the current frame or page. For example, on a title page, you may want to flow the text containing the document title into a frame you've named *Title*, then advance the flow to print the author's name into another frame you've named *Author*.

Because you need to use JCFrame methods when you want to render text and images independently from the flow, these methods are extremely important to the text flow of your layout. When you want to direct the flow through frames and pages, use JCFlow methods.

2.5.1 Frame Methods

Use JCFrame methods when you want to render content that is not part of the main flow. For example, the information contained in header and footer frames is rendered separately from the contents of the body frame. Until new instructions are given, information printed into the header frame of a template page will recur. For more information, refer to Headers and Footers, in Chapter 6.

Content is not added directly into the JCFrame's render list; rather, content is appended to a current line until a complete line is built or the current line is flushed. The current line always begins as zero height – no assumption is made about the attributes of the elements that will subsequently be added to it. Thus, the current line will always fit immediately following the just-completed previous line, even if there is virtually no space available at the end of a frame. As content is added to the current line, its size will be adjusted to fit the new elements. If an element is added to the line which makes the line too tall for the available space, then the frame will attempt a newColumn() action. In a case where the newColumn() action succeeds, the elements of the current line are mapped to their new positions and the flow continues. In the remainder of cases where there is no following column, the frame throws an EndOfFrameException. For more information on exceptions, please see Exceptions, in Chapter 5.

JCFrame methods cease rendering information when the content reaches the bottom of a frame, since they are not part of the flow. When the frame runs out of room, an EndOfFrameException is thrown.

JCFrame Method	Description
newColumn()	Generates a column break and advances the text to the next column in the specified frame. Throws an EndOfFrameException in the last (only) column of a frame.
newLine()	Ends the current line and transfers the flow to a new line. Throws an EndOfFrameException if there is not enough room to print the text.
print()	Renders the specified content to this frame. Throws an EndOfFrameException if there is not enough room to print the text.

2.5.2 Flow Methods

When a JCFlow object is instantiated for a document, it generates the document's first page. This first page will normally have one or more flow frames, and the first of these will be initialized as the current frame. Once a current frame has been initialized, all flow content is passed to that frame until the frame becomes full.

Note: If the first page does not have a flow frame, successive pages will be generated until a flow frame is discovered.

You create the flow by instantiating a JCFlow object, passing in a JCDocument as a parameter. The passed-in JCDocument is the only one that may be associated with the JCFlow object. Note that there can be only one JCDocument object per JCFlow; also, the JCDocument object must be the one passed to the JCFlow in the constructor.

The template page's FlowFrameList, FlowPageTemplate, and FlowSectionTemplate attributes control the order by which the flow progresses through frames and pages. To control flow beyond the standard sequence specified by the templates, the program needs to call JCFlow methods. The following table describes the methods used to control flow.

JCFlow Method	Description
print()	Renders the specified content to the flow.
newLine()	Ends the current line and begins a new line.
newParagraph()	Begins a new paragraph.
newColumn()	Advances the text flow to the top of the next column, in the next frame if necessary.

JCFlow Method	Description
`newFrame()`	Advances the text flow to the next frame, generating a new page if necessary.
`newPage()`	Creates a new page based on the current page's `FlowPageTemplate`, and directs the flow to the first frame of the new page's `FlowFrameList`.
`newSection()`	Creates a new page based on the current page's `FlowSectionTemplate`, and directs the flow to the first frame of the new page's `FlowFrameList`.

2.5.3 A Flow Programming Example

The following example of flow programming comes from *sample.java*, which you can find in the */examples/pagelayout/* folder of your JClass PageLayout installation directory.

```
// Flow simple text into the document
// while occasionally changing the text style.

flow.setCurrentTextStyle(normal);
normal.setFontStyle(Font.BOLD | Font.ITALIC);
flow.print("Hello, world!");
flow.newParagraph();
normal.setFontStyle(Font.PLAIN);

flow.print("This is a simple ");
normal.setFontStyle(Font.BOLD | Font.ITALIC);
flow.print("JClass PageLayout ");
normal.setFontStyle(Font.PLAIN);
flow.print("example which does a number of different things.");
flow.newParagraph();

// talk about headers and footers
flow.setCurrentTextStyle(heading);
flow.print("Headers and Footers");
flow.newParagraph();
flow.setCurrentTextStyle(normal);
flow.print("First off, what we have done is set up a simple page
   with a ");
normal.setFontStyle(Font.ITALIC);
flow.print("header");
normal.setFontStyle(Font.PLAIN);
flow.print(" and a ");
normal.setFontStyle(Font.ITALIC);
flow.print("footer");
normal.setFontStyle(Font.PLAIN);
flow.print(". The header contains a right justified title and the ");
flow.print("footer contains a centered macro that prints the
   current ");
flow.print("page number.");
```

2.5.4 Typical Content Flow Sequence

Several events are normally involved in adding content to the flow. These events are listed below.

1. The application declares the addition of a text String to the flow, calling `JCFlow.print (String)`.
2. `JCFlow.print()` passes the String and the current `JCTextStyle` to the corresponding `JCFrame.print(JCTextStyle, String)` method.
3. `JCFrame.print()` encapsulates the String object in a `StringRender` object, using the `JCTextStyle` to supply formatting information and font metrics information from the current Graphics (provided by the `JCPrinter`).
4. `JCFrame.flowPrint()` is called with the new `StringRender` object. `flowPrint()` determines the amount of available space (subject to tab position and alignment, indentation and margin) on the current line to see whether the String will fit. If the text is too long to fit, `flowPrint()` will attempt to split off part of the String to fit into the available space.
5. The text, or the portion of it that fits on the line, is added to the current line in memory, and any required adjustments are made to the height of the line and its baseline. (If the text is larger than previous elements on the line more space will be needed. Similarly, if the text is in superior subscript mode, then that may increase the height of the line.)
6. If the line has been made too high to fit in the amount of vertical space currently available, then the action is abandoned and the current line is stored in a `EndOfFrameException` object which is thrown to be caught by the `JCFlow` action.
7. If the line fits, but not all (or none) of the text was added to the line, the current line is pasted into the frame and a new line is begun. The remaining text is printed into the new line.
8. If an `EndOfFrameException` is thrown, it is caught in the `JCFlow.print()` method, which will then find the next flow frame and pass to it the current line of text which did not fit, followed by any other pending content.

Please note that the same sequence is followed for embedded objects, except that it is not possible to split them; thus, if embedded objects do not fit on the current line, they are handed in their entirety to the following line.

3

Formatting Text

Working With Text Styles ■ *Working with Fonts* ■ *Modifying Paragraphs* ■ *Inserting Tabs*

3.1 Working With Text Styles

The JCTextStyle class gives you control over the appearance of text in your document output. Many applications require several styles for different types of paragraphs, such as headings, addresses, indented block quotes, and so on. You can use any of the standard styles that come with JClass PageLayout, or you can create and modify your own styles.

3.1.1 Using Standard Styles

Although you can easily create and modify your own styles, you may want to take advantage of the built-in standard styles found in JCTextStyle. You can apply any standard style using JCFrame.print(), for example:

```
frame.print(JCTextStyle.HEADING_BOLD, "North America");
```

The preceding example prints the text “North America” in the standard style HEADING_BOLD. Standard styles are constants and, by convention, always appear in uppercase letters. You cannot modify the standard styles themselves, but you can use them to create your own styles. For more information, refer to Section 3.1.2, Creating and Modifying Styles.

The following table lists and describes the appearance of the standard styles available in JClass PageLayout.

Style	Appearance
BOLD	Left-aligned, single-spaced, 10 pt. bold Times New Roman.
BOLD_ITALIC	Left-aligned, single-spaced, 10 pt. bold, italic Times New Roman.
CODE	Left-aligned, single-spaced, 10 pt. plain Courier.
CODE_INDENTED	Left-aligned, single-spaced, 10 pt. plain Courier with left, right, and paragraph indents of 0.25”.
DEFAULT_HEADER	Center-aligned, single-spaced, 14 pt. bold Times New Roman.

Style	Appearance
DEFAULT_TEXT	Left-aligned, single-spaced, 12 pt. plain Times New Roman.
HEADING	Left-aligned, single-spaced, 10 pt. plain Helvetica.
HEADING_BOLD	Left-aligned, single-spaced, 10 pt. bold Helvetica.
HEADING1	Left-aligned, single-spaced, 18 pt. bold Helvetica.
HEADING2	Left-aligned, single-spaced, 18 pt. plain Helvetica.
HEADING3	Left-aligned, single-spaced, 16 pt. bold Helvetica.
HEADING4	Left-aligned, single-spaced, 16 pt. plain Helvetica.
HEADING5	Left-aligned, single-spaced, 14 pt. bold Helvetica.
HEADING6	Left-aligned, single-spaced, 14 pt. plain Helvetica.
HEADING7	Left-aligned, single-spaced, 12 pt. bold Helvetica.
INDENTED	Left-aligned, single-spaced, 10 pt. plain Times New Roman, with left, right, and paragraph indents of 0.25”.
ITALIC	Left-aligned, single-spaced, 10 pt. italic Times New Roman.
NORMAL	Left-aligned, single-spaced, 10 pt. plain Times New Roman.
PLAIN	Left-aligned, single-spaced, 10 pt. plain Times New Roman.

3.1.2 Creating and Modifying Styles

When you instantiate a JCDocument object, JClass PageLayout generates a default style (plain 12 pt TimesRoman) for any text you print. You can create and modify styles that control the appearance of text in your document, including font selection, indents, and line spacing.

A quick way to create a JCTextStyle object is to clone an existing one, saving you the trouble of specifying every attribute of the style.

```
JCTextStyle style = (JCTextStyle) JCTextStyle.NORMAL.clone();
style.setName("Body");
style.setLeftIndent(new JCUnit.Measure(JCUnit.CM, 0.5));
style.setRightIndent(new JCUnit.Measure(JCUnit.CM, 0.5));
style.setParagraphIndent(new JCUnit.Measure(JCUnit.CM, 0.5));
flow.setCurrentTextStyle(style);
```

The preceding example creates a JCTextStyle by cloning the standard NORMAL style, uses setName() to name it Body, and gives it left, right, and paragraph (first line) indents of 0.5 centimeters. JCFlow.setCurrentTextStyle() is called to apply this style to the text in the current flow. All subsequent text will appear in this style until a new style is applied.

3.2 Working with Fonts

Java maps fonts from their AWT names to their platform-specific equivalents. For example, "TimesRoman" maps to "Times New Roman" in Windows. When you program a font change, you use the AWT name. JClass PageLayout then identifies a PCL, PDF, or PostScript font that corresponds to the desired Java font. In this way, your code can run across platforms, finding and using the desired fonts.

In JClass PageLayout, you can specify which font a text style is to use, for example:

```
JCTextStyle.setFontFamily("TimesRoman");
```

or by passing in an actual Java font object, for example:

```
JCTextStyle.setFont(new java.awt.font("TimesRoman", Font.PLAIN, 12));
```

Using a font map, JClass PageLayout then identifies a PCL, PDF, or PostScript font that corresponds to the selected Java font.

You can use any Type 1 or Unicode mapped font in addition to the five fonts supported in JDK 1.1. These fonts have slightly different names on different platforms, so Java maps them to their platform-specific equivalents. Java 2 supports the five JDK 1.1 fonts, and also uses `java.awt.Font` to provide access to any font supported on your computer. For more information, refer to the next section.

If you want to use additional fonts, you must create a new font map and either an Adobe or PCL JAR file. For more information, refer to Section 3.2.2, Adding Font Metrics Files to JAR Files, and Section 3.2.3, Putting It Together. You also have the option of adding Unicode mapped TrueType font files (see Section 3.3, Adding Your Own Fonts for PDF Output for more information). JClass PageLayout can be programmed to embed TrueType fonts in the output file, resulting in a very portable document. If PDF output is not used, or if JClass PageLayout is not programmed to embed TrueType fonts in PDF files, you must verify that any fonts used in your document are available to the system's printer drivers.

3.2.1 Mapping Fonts

The font map uses `java.util.ListResourceBundle` to map between the platform-independent AWT font names you should use in your programs and the AFM, TFM and TTF font names JClass PageLayout uses internally. For example:

AWT	Internal
`{"TimesRoman",`	`"Times-Roman"},`
`{"TimesRoman-Bold",`	`"Times-Bold"},`
`{"TimesRoman-Italic",`	`"Times-Italic"},`

AWT	Internal
{"TimesRoman-BoldItalic",	"Times-BoldItalic"},

Since ListResourceBundle does not handle spaces in standard font names, the name must be specified internally by replacing spaces in the font name with underscores, as in this example for the font Courier New:

AWT	Internal
{"Courier_New",	"Courier"},
{"Courier_New-Bold",	"Courier-Bold"},
{"Courier_New-Italic",	"Courier-Oblique"},
{"Courier_New-BoldItalic",	"Courier-BoldOblique"},

See com.klg.jclass.page.adobe.JCAdobeFontMap for the default font mappings that come with JClass PageLayout.

Because ListResourceBundle does not allow underscores, JClass PageLayout removes them at runtime. So, when you request an AWT font, no underscores are required. For example, if you enter:

```
    JCTextStyle.setFontFamily("Courier New");
```

JClass PageLayout automatically substitutes the mapped Java font name.

You are able to instantiate a java.awt.Font with any name you choose, but to use a font other than the five supported by JDK 1.1 (Dialog, DialogInput, Monospaced, SansSerif, Serif) you must create a mapping from its font name to a format recognizable by JClass PageLayout. The easiest way to do this is to create a MyFontMap.properties file that contains the mapping.

For an example, please refer to Section 3.2.3, Putting It Together, later in this section, and to the FontMap example in the *examples/pagelayout* directory.

3.2.2 Adding Font Metrics Files to JAR Files

To use a font other than the five Java fonts, you must supply JClass PageLayout with the font metric information it needs to correctly render the text to the page. To do so, you must provide the Adobe or Hewlett-Packard font metrics file in a named JAR file. (The fonts that come with JClass PageLayout are archived in *fonts.jar*, part of the larger *jcpagelayout.jar*, found in the */lib/* subdirectory of your JClass PageLayout installation.)

Note: This does not apply for the use of TrueType font files in PDF output.

When you extract files from *jcpagelayout.jar*, you'll notice that there are separate *fonts.jar* files in the */page/adobe/* and */page/pcl/* subdirectories. Adobe font metrics files have an *.afm* extension. Hewlett-Packard PCL font metric files have a *.tfm* extension.

Adding a font is a three-step process:

1. Create a resource file to map the name by which it will be known in JClass PageLayout to its PostScript, PDF, or PCL printer name. Give the resource file any name you wish, but its extension must be *.properties*.
   Entries in this file look like this (comment lines begin with #):
   ```
   #   PageLayout          PostScript Font Name
       Optima=             Optima
   ```
   This file is required even if the alias names are the same as the printer names, as they are in the example above.
2. Provide a JAR file containing the font metrics (*.afm* or *.tfm* files). Adobe font metrics may be freely downloaded from Adobe's Web site.
3. Use a PCL, PDF, or PostScript printer constructor and pass it the location of the JAR file, the print file extension (*.afm* for PDF and PostScript, or *.tfm* for PCL), and the location of the font map resource file. Since the JAR file and the resource file are loaded as resources, they are specified using the CLASSPATH-relative dot notation, for example, `examples.pagelayout.MoreFontMappings`.

Your JClass PageLayout application is now able to use both the standard fonts stored in *fonts.jar* and the custom fonts you have just referenced.

Note: To use any font, you must have purchased a legitimate copy and installed it on your system. The AFM or TFM file contains a description of the font, not the font itself.

3.2.3 Putting It Together

For example, suppose you wanted to use the Adobe font Optima in a JClass PageLayout document. To do so, you would follow these steps:

1. Acquire the *glb_____.afm* file. If you do not have the file, you can download it from *ftp://ftp.adobe.com/pub/adobe/type/win/all/afmfiles/*.
2. Extract *fonts.jar* from *jcpagelayout.jar*, found in the */lib/* subdirectory of your JClass PageLayout installation.
3. Add *glb_____.afm* to *fonts.jar.*
4. Create a *MyFontMap.properties* file that maps the AWT font name to its AFM equivalent, for example:

   ```
   Optima = Optima
   ```

5. In your main document file, when instantiating the printer object, modify it to reflect the location of *MyFontMap.properties*, for example:
   ```
   JCPrinter printer = new JCPDFPrinter(
       System.out,
       new com.klg.jclass.page.adobe.postscript.AFMParser(),
       "/com/klg/jclass/page/adobe/fonts.jar", // jar location
       ".afm",                                 // file extension
       "examples/pagelayout/MyFontMap");        // user font map file
   ```
6. In your main document file, apply Optima to the current text style, for example:
   ```
   JCTextStyle.setFont(new java.awt.Font("Optima",
       Font. BOLD, 12));
   ```

3.2.4 Underlining

Underline mode is turned on and off by passing constants `JCTextStyle.LINEMODE_UNDERLINE` and `JCTextStyle.LINEMODE_NONE` to the `JCTextStyle` method called `setUnderlining()`. The following code fragment, taken from *examples/pagelayout/UnderlineExample.java*, turns underlining on, prints some underlined text, then returns the flow to normal, non-underlined mode:

```
// create a text style
JCTextStyle underlinedText = new JCTextStyle("Normal");
underlinedText.setUnderlining(JCTextStyle.LINEMODE_UNDERLINE);

// apply the style
flow.setCurrentTextStyle(underlinedText);

// print some text to the document
flow.print("This is some underlined text.");

underlinedText.setUnderlining(JCTextStyle.LINEMODE_NONE);

// apply the style
flow.setCurrentTextStyle(underlinedText);
```

3.2.5 Subscripts and Superscripts

The position of text relative to the baseline is controlled by `setBaselineOffset()`, which takes three parameters: `JCTextStyle.OFFSET_NONE`, `JCTextStyle.OFFSET_SUBSCRIPT`, or `JCTextStyle.OFFSET_SUPERSCRIPT`. Thus, to cause text to appear as a subscript, use

```
// Switch text style to subscript mode
style.setBaselineOffset(JCTextStyle.OFFSET_SUBSCRIPT);
```

to cause text to appear as a superscript, use

```
// Switch text style to superscript mode
style.setBaselineOffset(JCTextStyle.OFFSET_SUPERSCRIPT);
```

and to return text to normal, use

```
// Return text style to normal mode
style.setBaselineOffset(JCTextStyle.OFFSET_NONE);
```

To control the size of the subscripted or superscripted text, use the method called `setSubscriptRatio()`. It takes a double which specifies the size of the subscripted or superscripted text relative to the size of the current normally-sized text. If this method is not called, a default ratio of 0.75 is used.

3.2.6 Euro Symbol

In order to use the Euro symbol (€), first check whether the font you are using contains the Euro symbol.

The Euro character is supported for all output types except PCL.

If your font includes the Euro symbol

If the font you are using contains the Euro symbol, you can use this character in JClass PageLayout documents by printing the Unicode value of the Euro character (`U+20AC`) to a `JCFlow` or `JCFrame` object. For example:

```
flow.print("\u20ac");
```

The default fonts used by the Adobe Acrobat PDF Viewer (for instance, Helvetica, TimesRoman, Courier) and many browsers and printers now contain the Euro symbol. If you are using a custom font, ensure that the font contains the Euro symbol and that the symbol's metrics are included in the font's AFM file under the character name "Euro". For more details about working with custom fonts, please see Section 3.2.2, Adding Font Metrics Files to JAR Files.

To enable the Euro in PostScript printing, you must also set the `characterEncoding` property on the `JCPostScriptPrinter` class:

```
JCPostScriptPrinter printer;
printer.setCharacterEncoding
   (JCPostScriptPrint.ENCODING_ISO_LATIN_1_EURO);
```

Please note that the Euro symbol is the only Unicode character that JClass PageLayout supports for PCL and PostScript output formats.

If your font does not include the Euro symbol

If the font you are using does not contain the Euro symbol, there are two options with JClass PageLayout:

- make use of the Adobe Euro font package
- use a GIF file for the Euro symbol

The Adobe Euro font package contains free, downloadable font families comprising only the Euro symbol. Thus, when you want to use the Euro symbol, simply switch to one of the fonts in this package, print the Unicode value of the Euro character (`U+20AC`) to the desired `JCFlow` or `JCFrame` object, and then switch back to your previous font. The Adobe Euro font package is available as a free download from Adobe at *http://www.adobe.com/type/eurofont.html*

Alternatively, if the font you are using does not contain the Euro symbol, JClass PageLayout provides the Euro symbol as a GIF file (€). The *euro.gif* file is found in *pagelayout.jar* (*/com/klg/jclass/page/resources/*).

You can use the code in this sample to create and reference an `Image` object of the symbol and scale it to your desired size (for instance, to reflect the current point size of your text style).

Embedding the Euro GIF in text

To embed the Euro GIF in a line of text, call:

```
java.net.URL url = document.getClass().getResource("/com/klg/jclass/
       page/resources/euro.gif");
java.awt.Image euro = java.awt.Toolkit.getDefaultToolkit(). getImage(url)
```

Then, when you want to add the Euro symbol to a line of text, call:

```
flow.embedImage(euro, JCDrawStyle.POSITION_ON_BASELINE, new JCUnit.
       Dimension(.12, .12));
```

where the `JCUnit.Dimension` argument represents a suitable size for the `currentTextStyle`.

For more detailed information, please see the Euro example, automatically installed in *JCLASS_HOME/examples/pagelayout/*

3.3 Adding Your Own Fonts for PDF Output

To use a font other than the standard three fonts supported by PDF (TimesRoman, Helvetica, and Courier), you must supply JClass PageLayout with the information that it needs to render text to PDF correctly. If the font you are adding is a TrueType font, you must tell JClass PageLayout where the corresponding font program file (.TTF) is located. If the font you are adding is a Type 1 font, you must give JClass PageLayout a font metrics file (.AFM) that contains the metric information for the characters contained within the font.

Note that **JClass PageLayout does not embed Type 1 fonts** in its PDF output, so you must verify that any Type 1 fonts other than the standard three – TimesRoman, Helvetica, and Courier – are available on the system on which the PDF file is to be viewed or on the printer on which it is to be printed. **TrueType fonts may be embedded** in PDF output; please see Section 3.3.1, Setting TrueType Font Properties for more information.

To add your own fonts, follow these steps. These steps are based on the font example called *UnicodeExample.java*, which is automatically installed in your *JCLASS_HOME/examples/pagelayout/fonts/* directory when you install JClass PageLayout.

Note: If attempting to use a TrueType font on the Mac OS platform, please ensure that it is an actual Unicode mapped TrueType font, not just a sfat-housed font. Other snft-housed font varieties can be used on an Apple courtesy of Apple's Open Font Architecture, but as these may not be TrueType fonts (containing TrueType's required tables, they may not behave as expected when attempting to use them with JClass PageLayout).

1. **Locate the necessary files related to the desired new font.**

   To use a TrueType font, a legitimate copy of the font must be available on your system. The TTF file must first be found so that its location can be given to JClass PageLayout. On Windows platforms, most font files can be found in *C:\WINNT\FONTS* or *C:\WINDOWS\FONTS* (for Windows XP). On Unix systems, fonts might be found in */usr/lib/X11/fonts* or */usr/share/fonts*.

   To use a Type 1 font, an AFM file containing the metrics of the characters within the font is required. The AFM file contains a description of the font, but not the font itself. If your copy of the font did not come with an AFM file, you might find one on the Adobe Web site at *ftp://ftp.adobe.com/pub/adobe/type/*. For example, if you wanted to use the Adobe font Galliard in a JClass PageLayout document, you would acquire the font's AFM files from *ftp://ftp.adobe.com/pub/adobe/type/win/all/afmfiles/001-050/017/*.

2. **If JClass PageLayout does not recognize the name of the font you've added, you may need to create a *user.properties* font name map file.**

   You may need to create a file called *user.properties* that will specify a font name map. This file creates a mapping between the AWT font names that you will use in your program (for example, GalliardRoman) and the actual font name, as specified within the font itself (for example, Galliard-Roman). If the name of your font in its plain style does not contain any dashes or spaces, you may not need to create a *user.properties* file; an automatic mapping will be attempted by JClass PageLayout.

   Each line of the *user.properties* file consists of two names separated by an equals ("=") sign. The name on the left must be the AWT name, and the name on the right must be the actual font name found within the font file.

   - The AWT name (the name on the left) cannot contain spaces. Any spaces must either be escaped with a backslash ("\"), or replaced with underscores. Styled fonts must be listed in the file as *fontName-Style*, where *fontName* is the AWT font name (mentioned above), and *Style* is one of Bold, Italic, or BoldItalic.

     If you require a list of available AWT font names, call `java.awt.GraphicsEnvironment.getAllFonts()` to retrieve a list of fonts available on your system, and then call `getName()` on each font in the list.

   - The actual font name (the name on the right) must appear exactly as it does in the font file. Therefore, spaces must be left as they are. To determine the actual font names for True Type fonts, call `FontLibrary.getTTFFontNames(fontfileName)` on the font in question.

   Here is an example of the *user.properties* font name map file for a program that will use the GalliardRoman and CaslonRoman fonts. The name before the equals sign is the AWT font name and the name after the equals sign is the actual font name. Note how styled fonts have been specified and the way that spaces in the AWT font name have been dealt with.

```
GalliardRoman = Galliard-Roman
GalliardRoman-Bold = Galliard-Bold
GalliardRoman-Italic = Galliard-Italic
GalliardRoman-BoldItalic = Galliard-BoldItalic
Caslon\ Roman = CaslonRoman
```

3. **Tell JClass PageLayout where to find these files.**

   By calling `JCDocument.addFontPackage(String packagePath)`, where `packagePath` is a String representing the **absolute** path of a directory, all fonts represented by the TTF, TTC, or AFM files in the directory (and all subdirectories) will be added for use in JClass PageLayout and all font names from the *user.properties* file found in these same directories will be mapped. By default, JClass PageLayout lists the file system directories where system font files are normally kept:

- *C:\WINNT\Fonts* and *C:\WINDOWS\Fonts* for Windows systems
- */usr/share/fonts* and */usr/lib/X11/fonts* for UNIX and Macintosh systems

These directories can be scanned for fonts and font name map files automatically by calling `FontLibrary.setAutoLoad(true)` before the first use of `FontLibrary`, `JCDocument`, or `JCPDFPrinter`.

Fonts may also be added one at a time.

- The `FontLibrary.addFont(String fileLocation)` method adds a single font when passed an absolute path to a TTF, TTC, or AFM file.
- The method `FontLibrary.addFont(URL fontURL, int fontType)` loads a single font of the type specified in `fontType` from the passed URL object. `fontType` may be equal to either `AFM` or `TTF`.
- The method `FontLibrary.addRelativeFont(String fileLocation)` adds a single font when passed the location of a TTF, TTC, or AFM file relative to the CLASSPATH.

Font name map *.properties* files may also be added one at a time.

- The method `FontLibrary.addFontNameMap(String fileLocation, String name)` reads the font name map file found in the directory specified by the absolute path in `fileLocation` and whose name begins with `name` and ends with *.properties.*
- The method `FontLibrary.addFontNameMap(URL fontURL, String name)` reads the font name map file found in the directory specified by the passed URL object and whose name begins with `name` and ends with *.properties.*
- The method `FontLibrary.addFontNameMap(URL location)` reads the font name map found at the location specified by the passed URL object.
- The method `FontLibrary.addRelativeFontNameMap(String location, String name)` reads the font name map file found in the specified directory relative to the CLASSPATH and whose name begins with `name` and ends with *.properties.*

Note that once fonts and font name maps have been loaded, they are globally available to all programs running within the same class loader.

3.3.1 Setting TrueType Font Properties

There are some options available to users when using TrueType fonts with JClass PageLayout. These options include the ability to embed parts or the whole of a TrueType font within PDF output and the ability to specify character sets that may affect the size of PDF output.

These options are contained within the `TrueTypeFontProperties` class. An instance of the class that contains the values of properties for a particular font may be obtained by calling the method `FontLibrary.getTrueTypeFontProperties(String userFontName)`, where `userFontName` is the same AWT name of the TrueType font that is used to reference the font within the user program. If an instance of the class has not yet been associated with the specified font, one will be created.

The following properties are available in the `TrueTypeFontProperties` class. Values for each property may be set via a `set`*`PropertyName`*`()` method and retrieved via a `get`*`PropertyName`*`()` method.

Property Name	Description
`EmbeddingRules`	Specifies whether a TrueType font should be embedded within PDF output. A user may wish to embed a TrueType font if the desired font is not expected to be available on all systems on which the created PDF document will be viewed or printed. ■ A value of `DO_NOT_EMBED`, the default, will not embed the font. ■ A value of `EMBED_NEEDED` will embed only the characters from the font that are used within the document. ■ A value of `EMBED_ENTIRE_FONT` will embed the entire font program within the resulting PDF file.

Property Name	Description
`CharacterRange`	Specifies the range of font characters that will be used within the user's document. The property is used to determine which method should be used to encode text within the resulting PDF document. Most users will never need to set this property. ■ A value of `AUTO_DETECT`, the default, indicates that JClass PageLayout should automatically detect changes between the basic ANSI character set (character codes 0-255) and the larger Unicode character set (character codes 0-65535) and write out text in the format to which the character corresponds. ■ A value of `ANSI` will write out all characters in the single-byte ANSI format. This value should be used only if the user is certain that the document will never use a character outside the range 0-255. ■ A value of `UNICODE` will write out all characters as hexadecimal representations of the multi-byte Unicode format. Its use may result in larger PDF files than the other two settings in some cases, although it may result in a smaller PDF file if the majority of characters the user is referencing lie outside of the ANSI character set.
`IncludeUnicodeMap`	Specifies whether a `ToUnicode` character map will be included in PDF output. The default value is `true`. The `ToUnicode` character map is used by the PDF viewer program to translate between glyph codes and the characters they represent. It is necessary to allow functions such as cut-and-paste to work correctly. If such functionality is not needed by the user, setting this property to `false` will result in a smaller PDF file.

3.4 Modifying Paragraphs

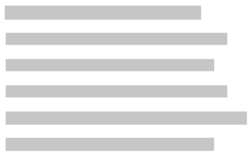
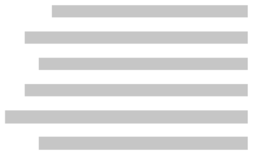
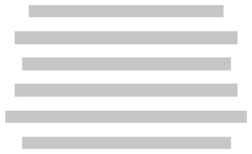
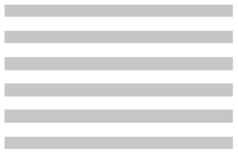
You can set a style's font properties to modify individual words or characters in a paragraph, but properties such as alignment, indents, line spacing, and paragraph spacing apply to the entire paragraph.

3.4.1 Alignment

JClass PageLayout supports the usual paragraph alignment options; left, center, right, and justified. Left alignment is the default.

To control the alignment of a paragraph style, call `JCTextStyle.setAlignment()`, for example:

```
    style.setAlignment(JCTextStyle.ALIGNMENT_CENTER);
```

Parameter	Result	
ALIGNMENT_LEFT	Paragraph text is left aligned.	
ALIGNMENT_RIGHT	Paragraph text is right aligned.	
ALIGNMENT_CENTER	Paragraph text is center aligned.	
ALIGNMENT_JUSTIFY	Paragraph text is left and right aligned.	

3.4.2 Indents

Indent properties control how far from the edge of the frame JClass PageLayout renders the text.

To control indentation, call the appropriate JCTextStyle indent method. To define the indentation width, use a JCUnit.Measure object. In turn, JCUnit.Measure requires JCUnit.UNITS to specify the appropriate units of measurement. For example:

```
    style.setLeftIndent(new JCUnit.Measure(JCUnit.INCHES, 0.25));
```

The preceding example creates a left indent at 0.25” from the edge of the frame.

JCTextStyle Method	Description
setLeftIndent()	Defines the amount of space between the left side of the frame and the left edge of every line in the paragraph except for the first line.
setParagraphIndent()	Defines the amount of space between the left side of the frame and the left edge of the first line in the paragraph.
setRightIndent()	Defines the amount of space between the right side of the frame and the right edge of the paragraph.

The following diagram demonstrates how left, paragraph, and right indents are applied.

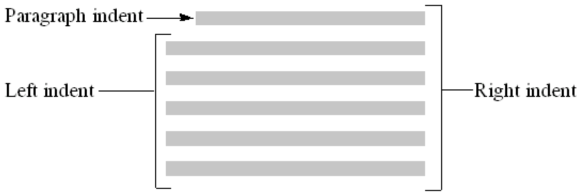


Figure 2 Different indentation types.

3.4.3 Line Spacing

Line spacing controls the amount of space between the baselines of text inside a paragraph. Line spacing is defined as a multiple of the height of a line of text in the current style.

```
    style.setLineSpacing(1.2);
```

The preceding example sets the line spacing of the current JCTextStyle object to 1.2 times the height of the text. If the text is 10 points high, 12 points of space are left between each line of rendered text.

3.4.4 Paragraph Spacing

Paragraph spacing controls the amount of space between paragraphs in the document. Paragraph spacing is measured between the baseline of the last line of text in the preceding paragraph and the baseline of the first line of text in the following paragraph. Note the difference from line spacing, which controls the amount of space between lines inside the paragraph. Like line spacing, paragraph spacing is defined as a multiple of the height of a line of text in the current style.

```
style.setParagraphSpacing(2);
```

If the text in the current style is 10 points high, JClass PageLayout leaves a gap of 10 points between paragraphs.

3.5 Inserting Tabs

You create a tab stop by creating a `JCTab` object. When doing so, you can define variables that control alignment, position, and fill.

The following example creates a left-aligned tab stop at the left margin with no fill.

```
tab = new JCTab();
```

You can instantiate a left-aligned Tab object at a position you specify:

```
tab = new JCTab(new JCUnit.Measure(JCUnit.CM, 3));
```

You can instantiate a Tab object, aligned and at a position you specify:

```
tab = new JCTab(new JCUnit.Measure(new JCUnit.CM, 8),
JCTab.TAB_ALIGNMENT_CENTER);
```

3.5.1 Adding Tabs to a Style

Typically, you set tab properties as part of a style, meaning that all further occurrences of that style are created with tabs in the same locations. To add tabs to a style, use `JCTextStyle.addTab()` or `setTabs()`.

To align to a defined tab, you must call the `JCFlow.tab()` method after adding your `JCTab` object to a style (that is, you need to call the `JCFlow.tab()` method in order to make use of the tab).

Method	Result
`addTab()`	Adds a tab to the style in the location specified.
`setTabs()`	Adds a list of identically aligned tabs to the style.

Using `addTab()`, you can add a single tab to the style, aligned and at a position you specify. The following example creates a left-aligned tab 3 cm from the edge of the frame.

```
JCTextStyle.addTab(JCTab.TAB_ALIGNMENT_LEFT,
new JCUnit.Measure(JCUnit.CM, 3));
```

Using setTabs(), you can add a list of regularly spaced tabs with a common alignment. The following example formats the style with eight left aligned tabs, each spaced one centimeter apart.

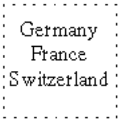
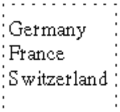
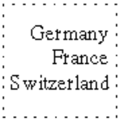
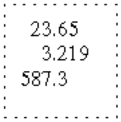
```
JCTextStyle.setTabs(JCTab.TAB_ALIGNMENT_LEFT,
new JCUnit.Measure(JCUnit.CM, 1), 8);
```

3.5.2 Tab Alignment

Once you have instantiated the JCTab object, you can use setTabAlignment() to adjust the way text is aligned to it. The following example aligns the right edge of the text to the tab stop location.

```
tab = new JCTab();
tab.setTabAlignment(JCTab.TAB_ALIGNMENT_RIGHT);
```

The following table describes the alignment options:

Field	Result	
TAB_ALIGNMENT_CENTER	Aligns an equal amount of text on either side of the tab stop.	 Germany France Switzerland
TAB_ALIGNMENT_LEFT	Aligns the left side (beginning) of the text with the tab stop.	 Germany France Switzerland
TAB_ALIGNMENT_RIGHT	Aligns the right side (end) of the text with the tab stop.	 Germany France Switzerland
TAB_ALIGNMENT_DECIMAL	Aligns a decimal or period (.) in the text with the tab stop. Primarily used for aligning columns of numbers.	 23.65 3.219 587.3

3.5.3 Tab Position

You use JCTab.setPosition() in combination with JCUnit.Measure() to adjust the horizontal (x-axis) location of the tab stop.

```
tab = new JCTab();
tab.setPosition(new JCUnit.Measure(JCUnit.INCHES, 1.0));
```

The preceding example creates a tab stop at 1”. You can use different units of measurement (POINTS or CM), if you prefer. If you do not declare a unit of measurement, JClass PageLayout uses the default unit type. For more information, refer to Units of Measurement, in Chapter 6.

3.5.4 Tab Fill

Often, users want to include a fill or leader between the text before and the text after the tab, as in the example of a Table of Contents:

4.4 Inserting Tab Stops 38

To control fill properties, use JCTab.setTabFill(). To create a right-aligned, right-margin tab filled with leader dots (such as the previous example), use the following code:

```
tab = new JCTab();
tab.setPosition(new JCUnit.Measure(JCUnit.INCHES, 5.5));
tab.setTabAlignment(JCTab.TAB_ALIGNMENT_RIGHT);
tab.setTabFill(JCTab.TAB_FILL_DOTS);
```

Other fill options are TAB_FILL_NONE and TAB_FILL_UNDERLINE.

4

Creating Tables

Tables in JClass PageLayout are implemented using the `JCPageTable` class. Tables contain column objects and row objects; the row objects contain the cells of the table. Table content is stored either in `JCFrame` objects belonging to individual cells or as objects in the entries of a `TableDataModel` associated with the table. Tables are printed by the flow, which breaks the table into rectangular ranges which are fit into flow frames.

4.1 Overview

The `JCPageTable` class provides methods and attributes for printing tables from your Java application. This chapter shows you how to:

- add a table to your document
- flow data into the table
- add a header row to the table
- add borders and background colors
- customize cell vertical alignment, margins, and borders
- span cells
- wrap tables
- convert table data from other Java sources

4.2 Table Structure

A table is created with a given number of columns, and a column object is created for each. The number of rows in a table is not fixed, and row objects are added to the table either explicitly, by using a method such as `JCPageTable.addRow()`, or implicitly, through reference to cell objects beyond the current last row of the table. The table's `rowList` attribute stores all the rows created within the table.

Cells of the table are stored in lists associated with particular rows. A row object with no cells containing content will have a cell list with no entries. As cells are created, the cell object will be stored at the correct position in the row's `cellList`. As with the table's `rowList`, blank (null) entries are created for non-existent cells. The content of a cell can be stored in a `JCFrame` object allocated by the cell; as a standard `JCFrame`, it can contain any content that is valid for a `JCFrame` object, up to and including another `JCPageTable`.

In addition to ordered lists of row and column objects, the table object can also (conditionally) own another `JCPageTable` object. The secondary `JCPageTable` object defines the headers of the parent table, but a header `JCPageTable` cannot own a header table. The structure of a header `JCPageTable` is identical to the structure of a main table, except that the column layout of the header table is defined implicitly by reference to the parent table.

There are also dummy cell objects attached to the table, and to each row and column object. These cell objects store column-, row-, and table-wide default cell attributes.

4.3 Using JCPageTable

Here is an overview of the general steps required to use `JCPageTable`.

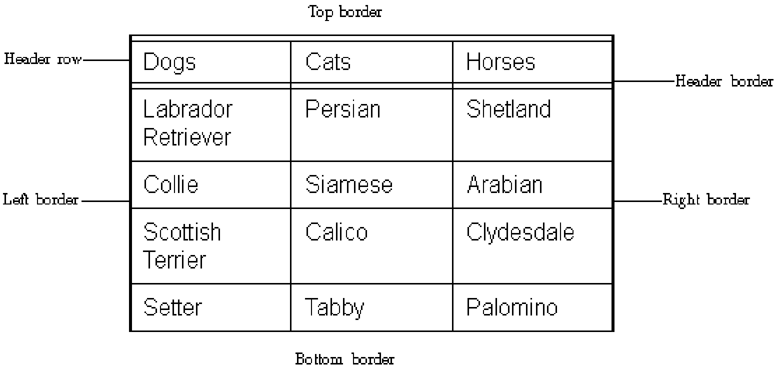
1. Create a `JCPageTable` object and pass in:

- the document to which this table belongs
- the number of columns in this table
- column widths

2. Create table header by calling `createHeaders()`, if desired.
3. Customize cells by setting borders and margins on table, rows, columns, or individual cells and by setting up any cell spans that are required.
4. Size table to frame using `fitToFrame()`, if desired.
5. Add data to tables (for example, call `JCPageTable.printToCell()` or `JCPageTable.addRow()`).
6. Print the table by calling `flow.print(table)`, where `flow` is an instance of `JCFlow`.

4.4 Creating a Table

By the end of this chapter, we will produce the following table:



Dogs	Cats	Horses
Labrador Retriever	Persian	Shetland
Collie	Siamese	Arabian
Scottish Terrier	Calico	Clydesdale
Setter	Tabby	Palomino

Top border
Header row
Header border
Left border
Right border
Bottom border

Figure 3   A Table example in JClass PageLayout.

To begin, we define a table with three columns, each of which is one inch wide. Here is the relevant code.

```
JCPageTable table = new JCPageTable(document, 3,
    new JCUnit.Measure(JCUnit.INCHES, 1.0));
```

The `JCPageTable` parameters indicate the `JCDocument` to which this table belongs and that this table should contain three columns, each one inch wide.

4.5 Adding Data to Tables

To print data into cells, use the `JCPageTable.printToCell()` method. It prints text to the cells you specify, generating them if necessary. For instance, the following code prints the text “Calico”, using the current `JCTextStyle` (the `style` in the code sample below), to the cell found in row `2`, column `1`.

```
table.printToCell(2, 1, style, "Calico");
```

Note: Cell indices start at (0, 0), independent of the presence of a header row.

4.5.1 Creating Body Rows

When you print to a cell frame that does not yet exist, you create a row of cells that correspond to the number of columns in the table. The following example populates the first three body rows of the table shown in Figure 3.

```
table.printToCell(0, 0, style, "Labrador");
table.printToCell(0, 1, style, "Persian");
table.printToCell(0, 2, style, "Shetland");
```

```
    table.printToCell(1, 0, style, "Collie");
    table.printToCell(1, 1, style, "Siamese");
    table.printToCell(1, 2, style, "Arabian");
    table.printToCell(2, 0, style, "Terrier");
    table.printToCell(2, 1, style, "Calico");
    table.printToCell(2, 2, style, "Clydesdale");
```

To actually print the table, call

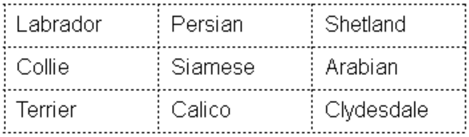
```
   flow.print(table);
```

where `flow` is an instance of `JCFlow`.

Note that JClass PageLayout identifies each cell by numbering cells in a `row, column` format, starting from zero (`0`).

Note that if a row cannot fit in the current page, then that row will be placed on the following page. However, if the height of that row is larger than the height of the entire page (for example, in the case that the row contains particularly extensive data, such as a long String), then that row is truncated.

The table is beginning to take shape:



Labrador	Persian	Shetland
Collie	Siamese	Arabian
Terrier	Calico	Clydesdale

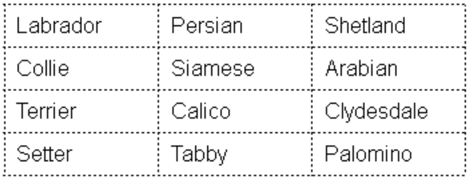
Figure 4 Table with body rows.

4.5.2 Adding Body Rows

You can also add rows of cells to the end of a table using `JCPageTable.addRow()`.

```
    table.addRow(style, newString[]{"Setter", "Tabby", "Palomino"});
```

The example produces the following results:



Labrador	Persian	Shetland
Collie	Siamese	Arabian
Terrier	Calico	Clydesdale
Setter	Tabby	Palomino

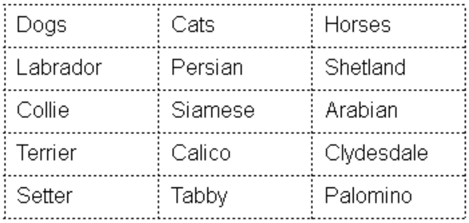
Figure 5 Table with additional body row.

4.5.3 Adding Header Rows

Suppose you wanted to create a table header row. Although a header row may look as though it is part of the table containing the body rows, in actuality it is a separate table unto itself. You build the header table using JCPageTable.createHeaders(). Once you have created the header table, you can populate its cells, much as you did the body rows.

```
JCPageTable header = table.createHeaders();
try {
JCFrame frame = header.getCellFrame(0, 0);
frame.print(style, "Dogs");

frame = header.getCellFrame(0, 1);
frame.print(style, "Cats");

frame = header.getCellFrame(0, 2);
frame.print(style, "Horses");
}
catch (EndOfFrameException e) {}
```

Adding the header row produces the following results:



Dogs	Cats	Horses
Labrador	Persian	Shetland
Collie	Siamese	Arabian
Terrier	Calico	Clydesdale
Setter	Tabby	Palomino

Figure 6 Table with header row.

4.6 Customizing Tables

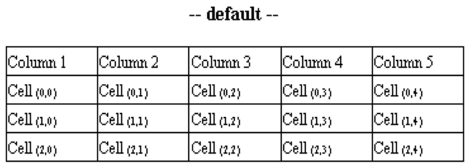
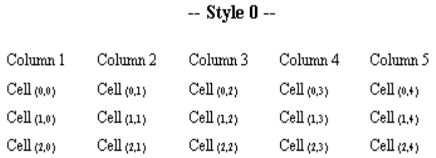
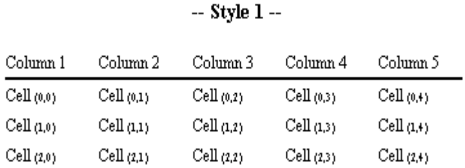
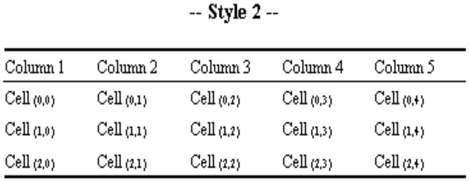
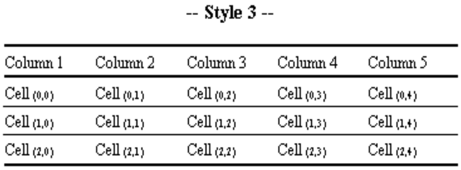
Now that you've laid out your table and printed data into it, you may want to customize its appearance by defining borders and background colors or by defining various line styles and shadings. This section describes how.

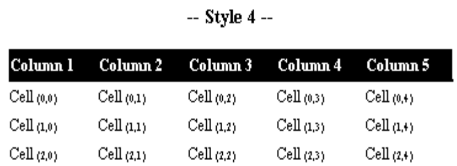
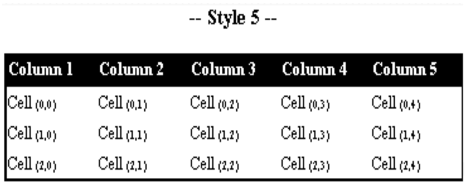
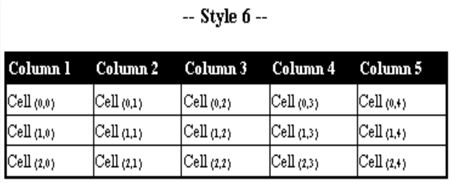
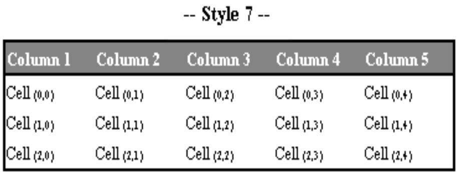
4.6.1 Table Styles

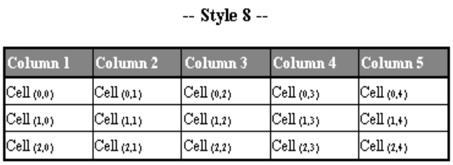
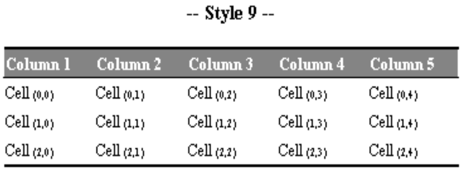
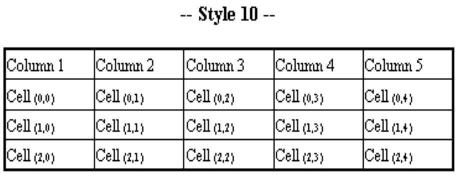
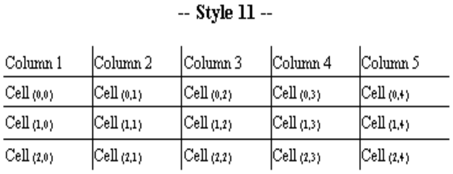
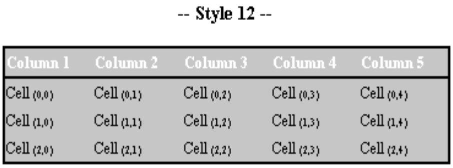
The simplest way to customize a table is to choose one of the numerous built-in table styles in JCTableStyle. Table styles are constants, such as JCTableStyle.STYLE_DEFAULT, or JCTableStyle.STYLE_*n*, where *n* ranges from 1 to 17.

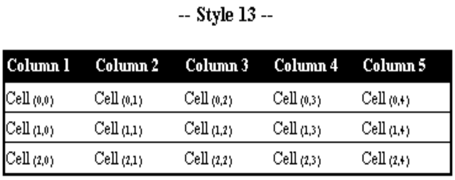
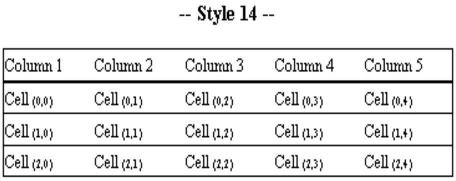
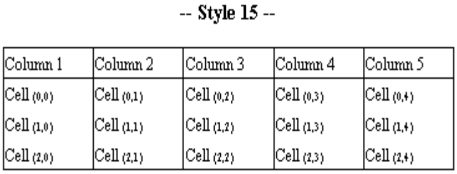
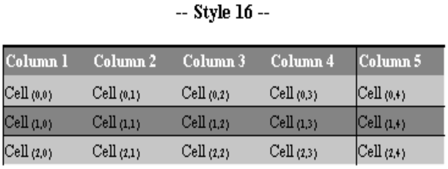
Table styles are cloneable. You can further customize a table style by cloning the one you wish to use as a base for making changes.

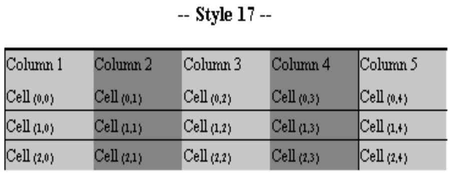
```
// Create a table style with orange headers and alternating colored
// rows
JCTableStyle tableStyle =
(JCTableStyle)JCTableStyle.STYLE_DEFAULT.clone();
tableStyle.getHeaderStyle().setBackground(Color.orange);
tableStyle.setAlternate( new JCAlternate(Color.lightGray,
Color.white, true));
```

<table>
<tr><th>Style Name</th><th>Description</th><th>Image</th></tr>
<tr><td><code>Default</code></td><td>- thin left, right, top, bottom, header, horizontal, and column borders
- no column shading
- regular heading font</td><td>
-- <b>default</b> --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td><code>Style0</code></td><td>- no left, right, top, bottom, header, horizontal, or column borders
- no column shading
- regular heading font</td><td>
-- <b>Style 0</b> --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td><code>Style 1</code></td><td>- thick header border; no other borders
- no column shading
- regular heading font</td><td>
-- <b>Style 1</b> --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td><code>Style 2</code></td><td>- thin header border; thick top and bottom borders; no other borders
- no column shading
- regular heading font</td><td>
-- <b>Style 2</b> --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td><code>Style 3</code></td><td>- thick top, bottom, and header borders; thin horizontal borders; no left, right, or column borders
- no column shading
- regular heading font</td><td>
-- <b>Style 3</b> --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
</table>

Style Name	Description	Image
`Style 4`	■ no left, right, top, bottom, horizontal, or column borders; thick header border ■ header colored (black); reverse type for header text ■ no column shading	 -- Style 4 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 5`	■ thick right, left, top, bottom, and header borders; no column or horizontal borders ■ header colored (black); reverse type for header text ■ no column shading	 -- Style 5 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 6`	■ thick right, left, bottom and header borders; thin horizontal and column borders; no top border ■ header colored (black); reverse type for header text ■ no column shading	 -- Style 6 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 7`	■ thick right, left, top, bottom and header borders; no horizontal or column borders ■ header colored (gray); reverse type for header text ■ no column shading	 -- Style 7 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)

<table>
<tr><th>Style Name</th><th>Description</th><th>Image</th></tr>
<tr><td>Style 8</td><td>■ thick right, left, top, bottom, and header borders; thin horizontal and column borders
■ header colored (gray); reverse type for header text
■ no column shading</td><td>
-- Style 8 --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td>Style 9</td><td>■ thin header border; thick top, and bottom borders; no right, left, horizontal, and column borders
■ header colored (gray); reverse type for header text
■ no column shading</td><td>
-- Style 9 --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td>Style 10</td><td>■ thick right, left, top, and bottom borders; thin header, horizontal, and column borders
■ regular heading font
■ no column shading</td><td>
-- Style 10 --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td>Style 11</td><td>■ no right, left, top, and bottom borders; thin header, horizontal, and column borders
■ plain headers
■ no column shading</td><td>
-- Style 11 --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
<tr><td>Style 12</td><td>■ thick right, left, top, bottom, and header borders; no horizontal or column borders
■ header colored (gray); reverse type for header text
■ gray column shading</td><td>
-- Style 12 --
<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table></td></tr>
</table>

Style Name	Description	Image
`Style 13`	■ thick right, left, bottom, and header borders; thin horizontal border; no top or column borders ■ header colored (black); reverse type for header text ■ no column shading	 -- Style 13 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 14`	■ thin right, left, top, bottom, and horizontal borders; thick header border; no column border ■ plain header ■ no column shading	 -- Style 14 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 15`	■ thin right, left, top, bottom, header, and column borders; no horizontal border ■ plain header ■ no column shading	 -- Style 15 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 16`	■ thin top, bottom, right, left and horizontal borders; thick header border; thin border between last two columns ■ header colored (dark gray); reverse type for header text ■ no column shading ■ alternate row shading (light gray and dark gray)	 -- Style 16 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)

Style Name	Description	Image
`Style 17`	- thin top, bottom, right, left, and horizontal borders; thick header border; thin border between last two columns - header reverse type - alternate column shading (light gray and dark gray)	 -- Style 17 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)

JCTableStyle Methods

Method	Description
`setAlternate()`	See Alternating row or column colors for details.
`setBackground()`	Sets the background color for a table.
`setBottomBorder()`	Sets the bottom perimeter border of a table.
`setColumnBorder()`	Sets the vertical border used by the internal table cells.
`setHeaderBorder()`	Sets the border between the header and the first row of the table.
`setHeaderStyle()`	Sets a `JCTableStyle` instance to be used as a table style for the header table.
`setLeftBorder()`	Sets the left perimeter border of a table.
`setName()`	Sets the name of the table style.
`setRightBorder()`	Sets the right perimeter border of a table.
`setRowBorder()`	Sets the default border drawn between table rows.
`setTextStyle()`	Sets the default text style to be used when drawing text in table cells.
`setTopBorder()`	Sets the top perimeter border of a table.

Alternating row or column colors

It is often desirable to shade every other row or column differently for easier reading. The `JCAlternate` class is used for this purpose, and the `setAlternate()` method in `JCTableStyle` takes an instance of `JCAlternate` to specify the alternating colors. There are two default styles, one for columns and one for rows.

To show rows alternating between gray and light gray, use `JCAlternate.ROW`. To show columns alternating between gray and light gray, use `JCAlternate.COLUMN`. To choose your own colors, create an instance of `JCAlternate` and pass its constructor the two colors you want, and a Boolean flag specifying whether the alternation is to take place over rows or over columns. If the flag is set to `true`, rows alternate in color, if the flag is set to `false`, columns alternate in color. For example, the following causes rows to alternate between red and blue:

```
tableStyle.setAlternate(new JCAlternate(Color.red, Color.blue, true));
```

Row/Column Dominance

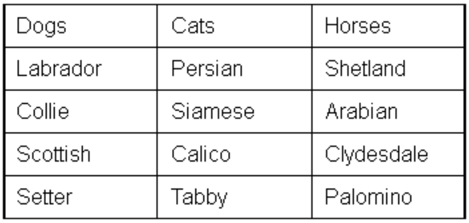
To control the order of border and cell color drawing, use the `RowColumnDominance` property of the table. A value of `ROW_DOMINANCE` forces column background colors and borders to draw before the equivalent properties of the table's row. In contrast, a value of `COLUMN_DOMINANCE` forces row background colors and border to draw before the equivalent properties of the table's columns. This property is set on the table class (`JCPageTable` or `JCFlowTable`) with the `setRowColumnDominance()` method.

4.6.2 Adding Borders

To define the appearance of a table border, you must select or create a `JCDrawStyle` with the required attributes and apply the style to the border. For information on controlling the appearance of borders, refer to Creating Draw Styles, in Chapter 6.

```
    table.setAllBorders(JCDrawStyle.LINE);
    table.setLeftBorder(JCDrawStyle.LINE_2POINT);
    table.setRightBorder(JCDrawStyle.LINE_2POINT);
```

This example uses `setAllBorders()` to apply the default single line style to all borders in the table, and then uses setLeftBorder and setRightBorder to apply thick side borders. If we were to apply this style to the table we developed earlier, the result would be as follows:



Dogs	Cats	Horses
Labrador	Persian	Shetland
Collie	Siamese	Arabian
Scottish	Calico	Clydesdale
Setter	Tabby	Palomino

Figure 7 Table with borders.

`JCDrawStyle` makes the following line styles available for table borders:

`LINE_TYPE_BROKEN`	Applies a dotted line to the table border.

LINE_TYPE_DOUBLE	Applies a double line to the table border.
LINE_TYPE_NONE	Blanks out the table border.
LINE_TYPE_SINGLE	Applies a single line to the table border.

JCPageTable provides numerous methods for applying line styles to table borders:

JCPageTable Method	Result
setAllBorders()	Applies the style to every border in the table.
setBottomBorder()	Applies the style to the bottom border of the table's last row.
setColumnBorder()	Applies the style to all column borders, except for the perimeter borders Left and Right.
setEdgeBorders()	Applies the style to all borders on the perimeter of the table (Top, Bottom, Left, and Right).
setHeaderBorder()	Applies the style to the border between the header table and the table's first row.
setHorizontalBorder()	Applies the style to the Top, Bottom, and header borders.
setInternalBorders()	Applies the style to all row and column borders except for the perimeter borders.
setLeftBorder()	Applies the style to the border on the table's left-hand side.
setRightBorder()	Applies the style to the border on the table's right-hand side.
setRowBorder()	Applies the style to all table row borders except for the header border and the perimeter borders Top and Bottom.
setTopBorder()	Applies the style to the top border of the table's first row. If there is a header table, applies the style to the top row of the header.
setVerticalBorders()	Applies the style to column, Left, and Right borders.

The behavior of perimeter borders at page breaks is determined by the BorderMode property of the table. A value of BORDER_USE_EXTERNAL forces perimeter borders to be used on each section of the multi-page table. A value of BORDER_USE_INTERNAL forces perimeter borders to be used only at the beginning and end of the table. Internal cell borders will be used for the table sections broken across pages. For example:

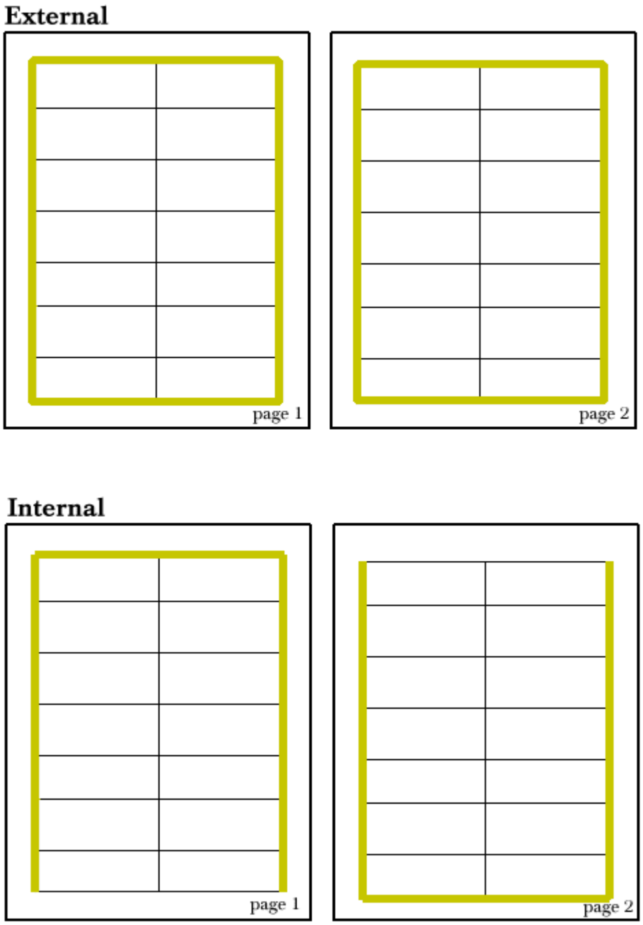


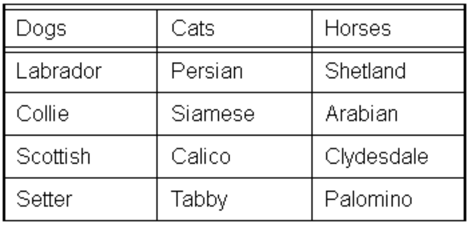
Figure 8 External and internal border behavior.

4.6.3 Adding Header Borders

As mentioned in Section 4.5.3, Adding Header Rows, the header row is a separate table unto itself. Modifying header borders is just like modifying borders in the main table. For example, to apply a double line to the top and bottom of the header row, enter:

```
table.setTopBorder(JCDrawStyle.LINE_DOUBLE);
table.setHeaderBorder(JCDrawStyle.LINE_DOUBLE);
```

The table appears as follows:



Dogs	Cats	Horses
Labrador	Persian	Shetland
Collie	Siamese	Arabian
Scottish	Calico	Clydesdale
Setter	Tabby	Palomino

Figure 9 Table with double header borders.

4.6.4 Applying Background Colors

You can further customize tables by applying background colors to cells, rows, columns, or the entire table. Colors and greyscale are defined by `java.awt.Color`. For more information on the colors and greys available to you, refer to the Java 1.2 API Specification at *http://java.sun.com/products/jdk/1.2/docs/api/index.html*.

For example, suppose you wanted to apply a green background to all of the cells in the first column of the table.

```
    table.getColumn(0).setBackgroundColor(Color.green);
```

This example uses the `getColumn()` method to identify the column to be colored, and then uses the `setBackgroundColor()` method to apply the specified static color.

4.6.5 Adjusting the Size of a Table

If you wish to adjust the size of a table so that it occupies all the space available in its parent frame, use the `fitToFrame()` method. The method takes two parameters: the frame to which the table should fit itself and the current `JCTextStyle`. It should only be called after all table borders, cell borders, and margins have been finalized.

4.7 Customizing Cells

`JCPageTable` has three inner classes: `JCPageTable.Cell`, `JCPageTable.Column`, and `JCPageTable.Row`, written to allow for more precise control over the description of individual table components.

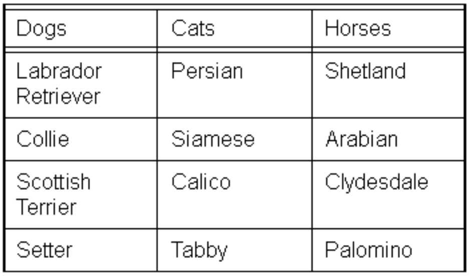
`JCPageTable.Column` and `JCPageTable.Row` provide methods that allow you to customize the cells in an entire column or row by modifying their alignment, background color, borders, and margins.

`JCPageTable.Cell` provides attributes and methods that allow you to customize cell appearance, including settings for vertical alignment, borders, margins, and spans.

4.7.1 Setting the Vertical Alignment

You can use the `JCPageTable.Cell.setCellAlignment()` method to adjust the vertical alignment of text in a cell.

Adding the words “Retriever” and “Scottish” to cells `0,0` and `2,0` increases the height of all of the cells in rows `0` and `2`. By default, the other cells in the same row align their text to the top of the cell.

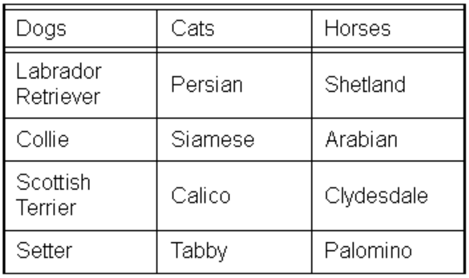


Dogs	Cats	Horses
Labrador Retriever	Persian	Shetland
Collie	Siamese	Arabian
Scottish Terrier	Calico	Clydesdale
Setter	Tabby	Palomino

Figure 10 Table cells with vertical alignment set to Top.

Suppose, for example, you want to vertically align that text to the middle of the cell. Here is the relevant code and how the altered table will appear:

```
table.setDefaultCellAlignment(JCPageTable.CELL_ALIGNMENT_CENTER);
```



Dogs	Cats	Horses
Labrador Retriever	Persian	Shetland
Collie	Siamese	Arabian
Scottish Terrier	Calico	Clydesdale
Setter	Tabby	Palomino

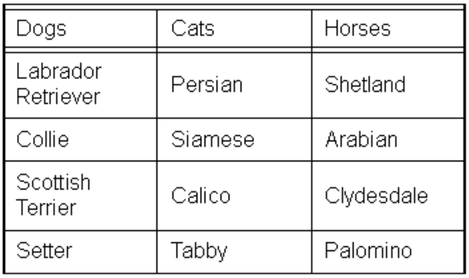
Figure 11 Table cells with vertical alignment set to Center.

You may also control the alignment in cells *individually*. The first three code snippets show how to align a cell to the bottom, top, and center of a cell, respectively. The last code snippet shows how to revert to the last alignment specified.

```
cell = table.getCell(0,1);
cell.setCellAlignment(JCPageTable.CELL_ALIGNMENT_BOTTOM);

cell = table.getCell(0,2);
cell.setCellAlignment(JCPageTable.CELL_ALIGNMENT_TOP);

cell = table.getCell(2,2);
```

```
    cell.setCellAlignment(JCPageTable.CELL_ALIGNMENT_CENTER);

    cell = table.getCell(2,1);
    cell.setCellAlignment(JCPageTable.CELL_ALIGNMENT_NONE );
```

`JCPageTable.getCell()` identifies the cells containing text we want to vertically re-align. `JCPageTable.Cell.setCellAlignment()` adjusts their vertical alignment in the cell, in this case, to the center of the cell. Our adjustments produce the following results:



Dogs	Cats	Horses
Labrador Retriever	Persian	Shetland
Collie	Siamese	Arabian
Scottish Terrier	Calico	Clydesdale
Setter	Tabby	Palomino

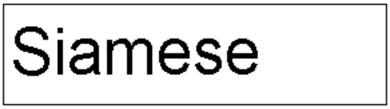
Figure 12 Table cells with vertical alignment set to Center.

`JCPageTable` provides the following cell vertical alignment options:

`CELL_ALIGNMENT_CENTER`	Aligns cell text to the middle of the row.
`CELL_ALIGNMENT_TOP`	Aligns cell text to the top of the row.
`CELL_ALIGNMENT_BOTTOM`	Aligns cell text to the bottom of the row.
`CELL_ALIGNMENT_NONE`	No specific cell alignment.

4.7.2 Defining Cell Margins

To control the space between the cell border and the text it contains, you adjust the cell margins. An enlarged version of one of the cells in the sample table provides a good example.



Siamese

Figure 13 Enlarged table cell.

The program has left gaps between the text and the left, right, top, and bottom borders of the cell. You can adjust the size of these gaps using JCPageTable.Cell methods.

```
cell = table.getCell(1,1);
cell.setBottomMargin(new JCUnit.Measure(JCUnit.POINTS, 8));
cell.setTopMargin(new JCUnit.Measure(JCUnit.POINTS, 8));
cell.setLeftMargin(new JCUnit.Measure(JCUnit.POINTS, 2));
cell.setRightMargin(new JCUnit.Measure(JCUnit.POINTS, 2));
```

The preceding example identifies the cell (1,1 – remember that the header row is numbered separately) for which margins are to be adjusted, and then calls the appropriate JCPageTable.Cell methods to set the top and bottom margins to 8 points and the left and right margins to 2 points.

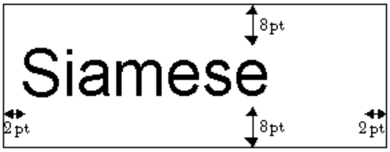


Figure 14 Enlarged table cell with margins illustrated.

4.7.3 Customizing Cell Borders

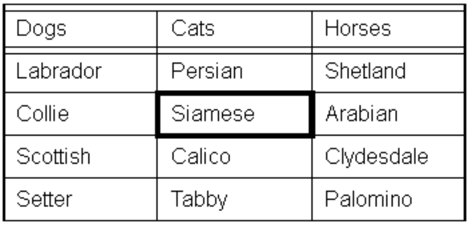
You can customize individual cells by overriding the border style specified by the JCDrawStyle and applied by the appropriate JCPageTable method. To override the style for a particular cell's borders, identify the cell, create a new JCDrawStyle, and apply it using a method from the JCPageTable.Cell class.

```
JCDrawStyle cellstyle = (JCDrawStyle)
JCDrawStyle.stringToStyle("default line").clone();
cellstyle.setLineType(JCDrawStyle.LINE_TYPE_DOUBLE);
cellstyle.setLineWidth(new JCUnit.Measure(JCUnit.POINTS, 1));
JCPageTable.Cell cell = table.getCell(1, 1);
cell.setBottomBorderStyle(cellstyle);
cell.setRightBorderStyle(cellstyle);
```

Note that style changes can also be made in one line as follows:

```
table.getCell(0, 1).setBottomBorderStyle(cellstyle);
table.getCell(1, 0).setRightBorderStyle(cellstyle);
```

After applying the above style changes to our table, this is the result:



Dogs	Cats	Horses
Labrador	Persian	Shetland
Collie	Siamese	Arabian
Scottish	Calico	Clydesdale
Setter	Tabby	Palomino

Figure 15 Table cell with customized borders.

In order to prevent cells from overriding each other's border styles, you can only specify the appearance of a cell's right and bottom borders. To achieve the results shown in the example, the program applies the `JCDrawStyle` to the right and bottom borders of the cell in question (Siamese), then applies the style to the bottom border of the cell immediately above it (Persian), and to the right border of the cell to its left (Collie).

4.7.4 Spanning Cells

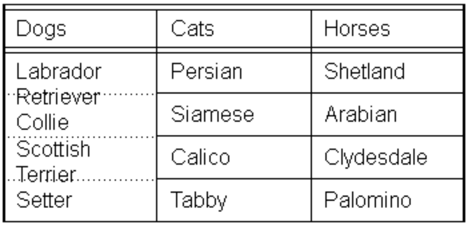
Spanning cells is the process by which the borders between specified cells are eliminated, creating one merged cell. The merged cell can cross multiple row and column borders, but does not alter the structure of the remaining cells, rows or columns in the table. Note that you should span cells before populating the cells.

To span cells, you pass to the `JCPageTable.spanCells()` method the row and column number of the cell that is the upper left-hand corner of the span, along with the number of rows to span down and the number of columns to span to the right. Follow this format:

```
table.spanCells(startRow, startColumn, numRows, numColumns);
```

For example, to span four cells down the first column of a table, beginning with the upper left-hand corner cell in the table, enter:

```
table.spanCells(0, 0, 4, 1);
```



Dogs	Cats	Horses
Labrador Retriever Collie Scottish Terrier Setter	Persian	Shetland
	Siamese	Arabian
	Calico	Clydesdale
	Tabby	Palomino

Figure 16 Table with spanned cells.

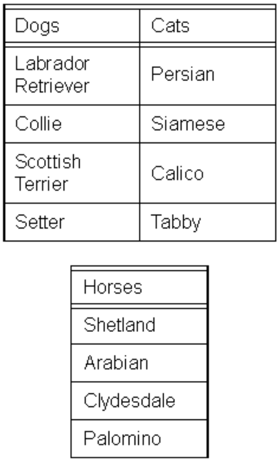
4.8 Table Wrapping

If a table is too long for the current frame, the extra rows flow to the next frame, along with the header row, which appears at the top of every frame to which the table rows flow.

If a table is too wide for the current frame, use `JCPageTable.setOverflowMode()` to handle the extra column data.

To wrap the extra column(s) so that they appear below the table:

```
table.setOverflowMode(JCPageTable.OVERFLOW_WRAP_COLUMNS);
```



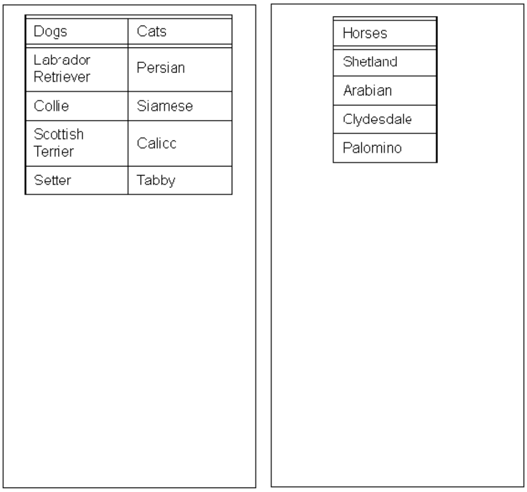
Dogs	Cats
Labrador Retriever	Persian
Collie	Siamese
Scottish Terrier	Calico
Setter	Tabby

Horses
Shetland
Arabian
Clydesdale
Palomino

Figure 17 Table with wrapped column.

To wrap the extra column(s) onto the following page:

```
table.setOverflowMode(JCPageTable.OVERFLOW_WRAP_COLUMNS_NEXT);
```



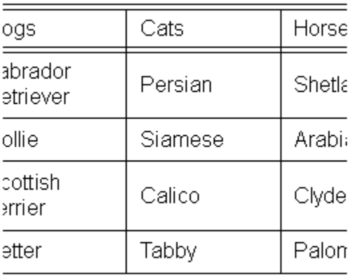
Dogs	Cats
Labrador Retriever	Persian
Collie	Siamese
Scottish Terrier	Calicc
Setter	Tabby

Horses
Shetland
Arabian
Clydesdale
Palomino

Figure 18 Table with column wrapped to the next page.

To clip an equal amount of data from the left and right-most columns in the table:

```
table.setOverflowMode(JCPageTable.OVERFLOW_CLIP_COLUMNS);
```



ogs	Cats	Horse
abrador etriever	Persian	Shetla
ollie	Siamese	Arabia
cottish errier	Calico	Clyde
etter	Tabby	Palon

Figure 19 Table with clipped columns.

4.9 Converting Tables

You can use JClass PageLayout to convert data from other Java table classes into `JCPageTable` tables. You can extract table data along with simple text formatting from JClass JCTables and Swing JTables. You can also extract the data from a JDBC result set and reconstruct it as a `JCPageTable`.

4.9.1 Converting JClass LiveTables

If you installed JClass LiveTable with your JClass DesktopViews installation, you can flow data from a JClass LiveTable `JCTable` into a `JCPageTable` in your JClass PageLayout application.

Creating a New JCPageTable

To flow `JCTable` data into a new instance of a JCPageTable, use `JCPageTableFromJTable.createTable()`. You have the following options:

Method	Description
`createTable(JCDocument doc, JCTable jcTable, boolean populate)`	If `populate` is `true`, creates a `JCPageTable` by duplicating all of the data and text formatting stored in the `JCTable`. If `populate` is `false`, an empty `JCPageTable` is created.
`createTable(JCDocument doc, TableDataModeltableDataModel, boolean populate)`	If `populate` is `true`, creates a `JCPageTable` by duplicating the data stored in the data source normally used by the `JCTable`. Since no `JCTable` is passed in, no text formatting is converted. If `populate` is `false`, an empty `JCPageTable` is created.

Populating an Existing JCPageTable

To flow `JCTable` data into an existing `JCPageTable`, use `JCPageTableFromJCTable.populateTable()`. You have the following options:

Method	Description
`populateTable(JCPageTable doc, JCTable jcTable)`	Populates a `JCPageTable` with all of the data and text formatting stored in the `JCTable`.
`populateTable(JCPageTable doc, TableDataModel tableDataModel)`	Populates a `JCPageTable` with the data stored in the data source normally used by a `JCTable`. Since no `JCTable` is passed in, no text formatting is converted.

4.9.2 Converting Swing JTables

You can also convert tables created with `javax.swing.JTable` into `JCPageTable` instances.

Creating a New JCPageTable

To create a new `JCPageTable` from an existing Swing `JTable` or `JTable` `TableModel`, use the `JCPageTableFromJTable.createTable()` method. You have the following options:

Method	Description

`createTable(JCDocument doc,`
`javax.swing.JTable jTable,`
`boolean populate)`
	If `populate` is `True`, creates a `JCPageTable` by duplicating all of the data, cell fonts, and text alignments from the view of the source `JTable`. If `populate` is `False`, an empty `JCPageTable`, with the same number of columns as the view of the source `JTable`, is created.

`createTable(JCDocument doc,`
`javax.swing.table.TableModel tableModel,`
`boolean populate)`
	If `populate` is `True`, creates a `JCPageTable` by duplicating all of the data from the source `TableModel`. No text formatting is performed. If `populate` is `False`, an empty `JCPageTable`, with the same number of columns as the source `TableModel`, is created.

Populating an Existing JCPageTable

To flow data and text styles from an existing Swing JTable or JTable TableModel into an existing JCPageTable, use the JCPageTableFromJTable.populateTable() method. You have the following options:

Method	Description
populateTable(JCPageTable table, javax.swing.JTable jTable, boolean applyJTableStyles)	Populates an existing JCPageTable with all of the data from the view of the source JTable. If applyJTableStyles is true, cell fonts and text alignments from the source JTable will be copied to the JCPageTable.
populateTable(JCPageTable table, javax.swing.table.TableModel tableModel)	Populates a JCPageTable with all of the data from the source TableModel. No text formatting is performed.

Examples

To perform a complete conversion of an existing JTable to a JClass PageLayout PageTable:

```
JCPageTable pageTable =
   JCPageTableFromJTable.create(document, jTable, true);
```

To convert an existing JTable to a JClass PageLayout PageTable, without preserving text formatting from the existing JTable so that new formatting can be applied:

```
// create JCPageTable with same number of columns as JTable
JCPageTable pageTable =
   JCPageTableFromJTable.create(document, jTable, false);
// apply our own text formatting to headers and body
JCPageTable headerTable = pageTable.getHeaders();
headerTable.getRow(0).setDefaultStyle(JCTextStyle.HEADING5);
pageTable.setDefaultStyle(JCTextStyle.CODE);
// populate JCPageTable with data from JTable
JCPageTableFromJTable.populate(pageTable, jTable, false);
```

4.9.3 Converting JDBC Databases

JDBC (Java DataBase Connectivity) is the part of the Java API (java.sql) that allows you to send SQL queries to a database. You can format the result set of an SQL query into a JCPageTable.

1. **Create the `JCPageTable` and the `ResultSet`.**

   ```
   protected JCPageTable createTable(JCDocument doc) {
   ResultSet resultSet = null;
   JCPageTable table= null;
   ```

2. **Load the JDBC driver.**

   ```
   try {
   Class.forName(driver); // Example: sun.jdbc.odbc.JdbcOdbcDriver
   } catch (ClassNotFoundException cnfe) {
   cnfe.printStackTrace();
   }
   ```

3. **Conduct the SQL query and return the `JCPageTable`.**

```
    try {
    Connection connection = DriverManager.getConnection(url,
                                                 login, password);
    Statement statement = connection.createStatement();
    resultSet = statement.executeQuery(query);
    table = com.klg.jclass.page.JCPageTableFromJDBC.createTable(doc,
        resultSet);

    // clean up
    resultSet.close();
    statement.close();
    connection.close();

} catch (SQLException sqle) {
    JOptionPane.showMessageDialog(null, sqle.toString(), "SQL error",
        JOptionPane.ERROR_MESSAGE);
    return null;
}

// return the JCPageTable
return table;
```

5

Adding Formulas to JClass PageLayout

5.1 Introduction

The formulae package in *com.klg.jclass.util* has special capabilities for working with mathematical objects. Similar to the way that objects such as `java.lang.Double` wrap a primitive type, those in *com.klg.jclass.util.formulae* encapsulate mathematical expressions (operators) whose operands may be scalars, vectors (in the mathematical sense), and matrices. These objects may then be stored as the generalized values of cells in a JClass PageLayout table, or in a JClass LiveTable, where they may be evaluated at run time to produce results based on the then-current data.

In addition, subclasses of `MathValue`, which are wrappers for generalized scalars, vectors, and matrices, provide several methods for converting an expression to a value and to a String, as well as other methods useful when dealing with these objects.

5.2 util.formulae's Hierarchy

The interfaces, abstract classes, and derived classes, including possible exception classes, are shown in Figure 20.

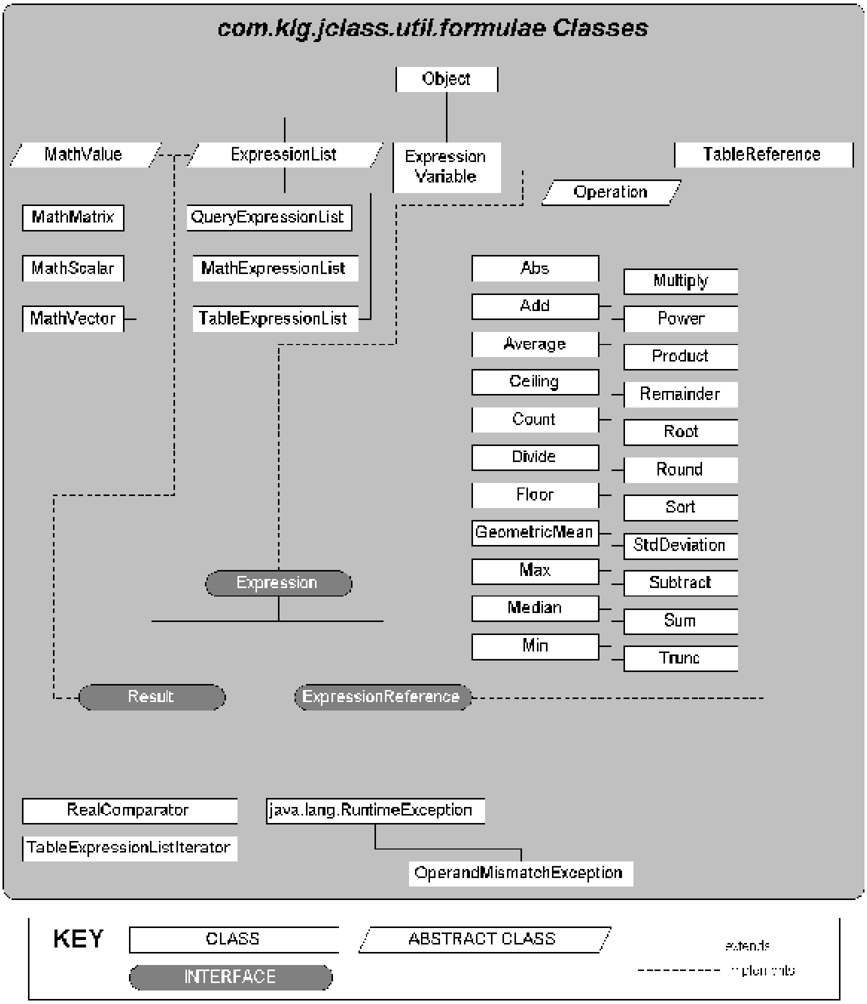


Figure 20   The inheritance hierarchy for com.klg.jclass.util.formulae.

The diverse set of mathematical operations permit you to compose complex mathematical formulas and provide references to them.

5.3 Expressions and Results

The top-level interface for the *com.klg.jclass.util.formulae* package is Expression, whose sole method is evaluate(). Any object that functions as an expression must have an evaluate() method that knows how to operate on data that might be a scalar, a vector, or a matrix. Applying the evaluate() method to an Expression produces a Result, which is a marker interface that identifies Expression types which are valid return types from the evaluation of other Expressions.

An Expression may be an Operation, as in

```
Expression f = new Add(op1, op2);
```

which, after evaluation, returns a Result.

5.4 Math Values

The abstract class MathValue forms the root for all derived constant-based result/data classes. It satisfies the Expression interface by defining an evaluate() method, which simply returns the MathValue as a Result. Its concrete subclasses are MathMatrix, MathScalar, and MathVector. Because MathValue has an evaluate() method it is an Expression. Thus, MathValues may be passed as Expression objects.

MathValue Methods

MathValue Method	Description
evaluate()	Satisfies the Expression interface by returning the stored value. No evaluation is required because no operation is implied.
getDataFormat()	Retrieves the NumberFormat associated with this data.
matrixValue()	Gets the contents of this MathValue as a matrix of Numbers.
numberValue()	Gets the contents of this MathValue as a Number.
setDataFormat()	Sets a NumberFormat to use on the contents of this MathValue.
vectorValue()	Gets the contents of this MathValue as a vector of Numbers.

Note: The subclasses of MathValue override all but the first method. Since, for example, matrixValue() is not appropriate to a MathScalar, it throws an UnsupportedOperationException if it is called. Other method-data type mismatches also throw UnsupportedOperationExceptions. The method tables for the subclasses indicate which methods are data type mismatches for the given class.

5.4.1 MathScalar

MathScalar is a scalar constant represented as a MathValue. By encapsulating it in this fashion it can support integer and real numbers, and it can be extended if necessary to support other types of scalar numbers. Its data field is a realValue, a Number that is output based on the current dataFormat kept in MathValue.

Example:

```
double s1 = 10.0; MathValue ss1 = new MathScalar(s1);
```

MathScalar Constructors

The no-argument constructor MathScalar() creates an instance that encapsulates the value 0.0, while the other three constructors take a double, an int, or a java.lang.Number argument.

MathScalar Methods

MathScalar Method	Description
matrixValue()	Throws an UnsupportedOperationException.
numberValue()	Gets the contents of this MathValue as a Number.
toString()	Returns a String representation of this value.
vectorValue()	Throws an UnsupportedOperationException.

5.4.2 MathVector

MathVector is a representation of the class of vectors in a linear algebra sense. They may also be used as operands in matrix multiplication. A MathVector encapsulates a list of values which may be integers, doubles, or more generally, objects of type Number. It has methods for retrieval or modification of a value at a particular index, and for outputting the list as a String. The operators discussed in the next section accept these objects as operands. For example:

```
double[] ed = {2.71828, 3.1415927, 1.6020505};
MathValue mv = new MathVector(ed);
```

MathVector Constructors

The constructors for MathVector parallel those for MathScalar, except they take arrays as parameters rather than single values.

MathVector Methods

MathVector Method	Description
getValueAt()	Retrieves the value at a particular index in the vector.
matrixValue()	Throws an UnsupportedOperationException.
numberValue()	Throws an UnsupportedOperationException.
setValueAt()	Sets the value at a particular index in the vector.
toString()	Outputs the value of this vector as a String.
vectorValue()	Gets the contents of this MathValue as a vector of Numbers.

5.4.3 MathMatrix

MathMatrix is a representation of the class of matrices, again in the sense of linear algebra. The package implements the basic addition and multiplication operations in matrix algebra, including multiplying a matrix by a vector. It has methods for retrieval or modification of a value at a particular pair of indices, and for outputting the matrix as a String. The operators discussed in the next section accept these objects as operands. For example:

```
double[][] m1 = {{1.1, 1.2, 1.3},
                 {2.1, 2.2, 2.3},
                 {3.1, 3.2, 3.3}};
MathValue mm = new MathMatrix(m1);
```

MathMatrix Constructors

The constructors for MathMatrix parallel those for MathScalar, except they take 2D arrays as parameters rather than single values.

MathMatrix Methods

MathMatrix Method	Description
getValueAt()	Retrieves the value at a particular row, column pair of index values in the matrix.
matrixValue()	Gets the contents of this MathValue as an array of Numbers.
numberValue()	Throws an UnsupportedOperationException.
setValueAt()	Sets the value at a particular row, column pair of index values in the matrix.
toString()	Outputs the value of this vector as a String.
vectorValue()	Throws an UnsupportedOperationException.

5.5 Operations

The abstract Operation class defines the basic elements of an operator. Binary operators have a left and right operand, which enables the correct ordering to be applied to matrix operations and any other non-commutative operators. Unary operators have a single operand. For example:

```
double[] ed = {2.71828, 3.1415927, 1.6020505};
double[] rd = {(Math.sqrt(5.0) + 1.0) / 2.0, 4.0, 32.0};

MathValue e = new MathVector(ed);
MathValue r = new MathVector(rd);

Expression add = new Add(e, r);
```

Operation Constructors

There is a no-argument constructor that creates a generic operator, and there are constructors for every unary and binary permutation of Expressions and Numbers. A sample constructor is: Operation(Expression left, Expression right).

Operation Methods

Operation Method	Description
`clone()`	Returns a deep-copy clone of the operation and of all operands.
`evaluate()`	Returns a `Result` containing the evaluation of the expression.

5.5.1 The Defined Mathematical Operations

Unary Operators

Unary operators take one parameter, which is either an `Expression` or a `Number`. Because they are `Expression`s they all have an `evaluate()` method which returns a `Result`.

Operator	Description
`Abs`	The class for the absolute value operation. The operand may be a `Number` or an `Expression`, which may be a `MathScalar` or an `ExpressionList`, but not a vector or a matrix.
`Ceiling`	Ceiling is defined as the least integer greater than or equal to the operand, which may be a MathValue.
`Floor`	Floor is defined as the greatest integer less than or equal to the operand.
`Root`	Returns the positive square root of its operand.
`Round`	Round is defined as nearest integer to the operand. Rounding is done to an even number if the operand is exactly midway between two integers.
`Trunc`	Takes the integer part of a number. Equivalent to rounding to the nearest integer closer to zero. Example: trunc(-3.5) = -3.

Binary Operators

Binary operators take two parameters, which are either `Expression`s or a `Number`s. Because they are `Expression`s they all have an `evaluate()` method which returns a `Result`,

Operator	Description
`Add`	Adds two `Expression`s. If the `Expression`s are vectors of the same length, pairwise addition is performed. Matrices may be added providing the two operands have the name number of rows and columns. Unary addition is possible, and returns the evaluated operand.

Operator	Description
Average	Average (arithmetic mean) is defined as the sum of all elements divided by the number of elements. Its one-parameter constructor is an Expression, usually a list. Its two-parameter constructors are combinations of Expressions and Numbers.
Count	Count determines the total number of elements in its operands. Its one- and two-parameter constructors take one or two Expressions (usually a list or lists) and count their elements.
Divide	Division is the ratio of two operands. The left operand is the numerator and the right operand is the denominator.
GeometricMean	Geometric mean is defined as the *n*th root of the product of a set of *n* numbers. Its one-parameter constructor takes an Expression, usually a list. Its two-parameter constructors take combinations of Expressions and Numbers, multiplying all elements together and taking the *n*th root.
Max	Max is defined for a pair of elements or across a list. It selects the largest element. Its one-parameter constructor takes an Expression, usually a list. Its two-parameter constructors take combinations of Expressions and Numbers, examining all elements and selecting the largest.
Median	The Median of a list is the middle element of a sorted list, or the average of the two middle values if the list has an even number of elements. Its one- and two-parameter constructors take one or two Expressions.
Min	Min is defined for a pair of elements or across a list. It selects the smallest element. Its one-parameter constructor takes an Expression, usually a list. Its two-parameter constructors take combinations of Expressions and Numbers, examining all elements and selecting the smallest.
Multiply	Multiplication is the product of a pair of elements. Its two-parameter constructors take combinations of Expressions and Numbers, examining all elements and selecting the smallest.
Power	The exponentiation (^) operation. Its two-parameter constructors take combinations of Expressions and Numbers. The left operand is the base and the right operand is the exponent.

Operator	Description
`Product`	A product can be performed on a pair of elements or across a list. The product of an ExpressionList is the product of its individual members. Multiplication order is left-to-right, and first element of a list to last element. The result of a matrix multiplication may depend on the order of the operands.
`Sort`	This operation returns a sorted list of the given elements. Any secondary or nested lists are flattened.
`StdDeviation`	The sample standard deviation, given by sd = root((sum(1 to *n*)(element - average)^2) / (n - 1)), where *n* is the number of samples and average is the sample average. It has one- and two-parameter constructors consisting of `Expression`s.
`Subtract`	The difference between two numbers. It has two-parameter constructors that take combinations of `Expression`s and `Number`s.
`Sum`	A sum can be performed on a pair of elements or across a list. Its two-parameter constructors take combinations of `Expression`s and `Number`s. Its one-parameter constructor usually takes an `ExpressionList`.

5.5.2 Reducing Operations to Values

Since `Operation`s are `Expression`s, they all have an `evaluate()` method. Evaluation returns a `Result`, which may be converted to a String for printing. Here is an example:

```
double edd = 2.0;
double exp = 8.0;
MathValue eddy = new MathScalar(edd);
MathScalar expy = new MathScalar(exp);

double[] ed = {2.71828, 3.1415927, 1.6020505};
MathValue e = new MathVector(ed);

    Expression pow = new Power(eddy, expy);
    Expression powr = pow.evaluate();
    // Either one of these has a toString() method
    System.out.println("Power without evaluate(): " + pow);
    System.out.println("Power with    evaluate(): " + powr);
```

After which the following is written on the output:

```
Power without evaluate(): com.klg.jclass.util.formulae.Power@eb4f3b8c
Power with    evaluate(): 256.0
```

You see that calling `evaluate()` is necessary to have a value returned by the implicit `toString()` call.

5.6 Expression Lists

Expression lists are handy containers that permit you to perform an operation on a group of values.

MathExpressionList

The example shown here uses the binary form of Add to find the grand total of all the elements in two ExpressionLists.

```
// Expression Lists
Expression[] exprs1 = {null, null, null, null, null, null, null,
                                              null, null, null};
for (int i = 0; i < 10; i++){
    exprs1[i] = new MathScalar(95 + i);
}
ExpressionList explist1 = new MathExpressionList(exprs1);

Expression[] exprs2 = {null, null, null, null, null, null, null,
                                              null, null, null};
for (int i = 0; i < 10; i++){
    exprs2[i] = new MathScalar(95 + i);
}
ExpressionList explist2 = new MathExpressionList(exprs2);

sss1 = new Sum(explist1, explist2);
ssss1 = sss1.evaluate();
System.out.println(
    "Summing ExpressionLists with    evaluate(): " + ssss1);
```

Here's the output:

```
Summing ExpressionLists with    evaluate(): 1990
```

QueryExpressionList

A QueryExpressionList is designed as a wrapper for a set of Expressions stored in a JDBC-type ResultSet, that is, the result of a database query. Users of JClass DataSource may also use this facility.

TableExpressionList

Expression lists may be used to extend data from portions of a JCPageTable to produce summary reports. See Section 5.8, Using Formulas in JClass PageLayout, for details.

5.7 Exceptions

OperandMismatchException

Various operations such as adding a number to a vector are not defined, whereas other operations such as multiplying a vector by a number can be interpreted as a scaling operation. At compile time numbers, vectors, and matrices can be declared as generic Expressions, making it impossible to predetermine which operations are not permitted.

A run time check of the validity of an operation must be made. If a mathematical construct is evaluated and found to be illegal, the class throws an OperandMismatchException.

ClassCastException

There are cases where a run time class cast exception may occur. While most of these should be avoidable by selecting the correct class (such as using Product rather than Multiply when multiplying two vectors) the fact that both take Expressions as their parameters makes it difficult to avoid the possibility of an end user passing in an incorrect type if your application permits flexible user input. You may permit substitution of one arithmetic class for another, since they are all Operations. This also opens the door to class cast exceptions.

If the possibility exists for either of these exceptions, your code should attempt to handle it.

5.8 Using Formulas in JClass PageLayout

5.8.1 Performing a Mathematical Operation on a Range of Cells

Expression Lists and Expression References

Expression list objects hold a group of Expressions. ExpressionList is an abstract class whose methods permit the inclusion of additional elements to those already present, a method for removing elements or clearing all elements, for retrieving an element, and for comparing with another list. These operations are common to the concrete classes MathExpressionList, QueryExpressionList, and TableExpressionList.

Expression lists may be used as arguments for all mathematical operations. When given an expression list, evaluating a unary operator such as Abs returns a list containing the absolute values of its input list. Binary operators may return a single result or a list. Given expression lists, the mathematical operators Abs, Add, Ceiling, Divide, Floor, Multiply, Power, Remainder, Root, Round, Sort, and Subtract return lists, while Average, Count, GeometricMean, Max, Median, Min, Product, and Sum all return a single result after evaluate() has been called on them.

Use TableExpressionList to perform an operation over a range of cells in a JCPageTable. The following code snippet shows that the required parameters are a table data model and a block of cells.

```
Expression expression = new TableExpressionList(
                        pageTable.getTableData(),
                        new MathScalar(startRow),   // first row
                        new MathScalar(endRow),     // last row
                        new MathScalar(startColumn), // first column
                        new MathScalar(endColumn)   // last column
                        );
Sum sum = new Sum(expression);
```

The next code fragment places the formula for the sum in a selected cell. The formula is evaluated and the value of the sum is written to the designated cell.

```
pageTable.getCell(i, j).setCellValue(sum);
```

The advantage of using TableExpressionLists is that the formulas containing them can be evaluated after all table data has been filled in.

6

Refining a Document

6.1 Headers and Footers

You create headers and footers in the page template by defining frames not connected to the document's main flow. Building Page Templates, in Chapter 2, shows you how to use JClass PageLayout's default templates, and lists the XML elements you can use to write your own templates.

Because header and footer frames are not connected to the main flow of the document, but are defined separately, any text or images you render to those frames in a page template are replicated on every page that is based on that template. For an example, refer to Section 6.3, Page Numbers.

The following XML template lays out a standard 8.5x11 page consisting of a header frame, a body frame, and a footer frame.

```
<PAGE NAME="BookLeft" UNIT="inches">
<LOCATION X="0" Y="0"/>
<SIZE WIDTH="8.5" HEIGHT="11.0"/>
<FRAME NAME="header">
<LOCATION X="0.5" Y="0.5"/>
<SIZE WIDTH="7.5" HEIGHT="0.5"/>
</FRAME>
<FRAME NAME="body">
<LOCATION X="0.5" Y="1.25"/>
<SIZE WIDTH="7.5" HEIGHT="8.5"/>
</FRAME>
<FRAME NAME="footer">
<LOCATION X="0.5" Y="10"/>
<SIZE WIDTH="7.5" HEIGHT="0.5"/>
</FRAME>
<FLOWFRAME NAME="body"/>
<FLOWPAGE NAME="BookRight"/>
<FLOWSECTION NAME="BookChapter"/>
</PAGE>
```

This page template defines the following layout.

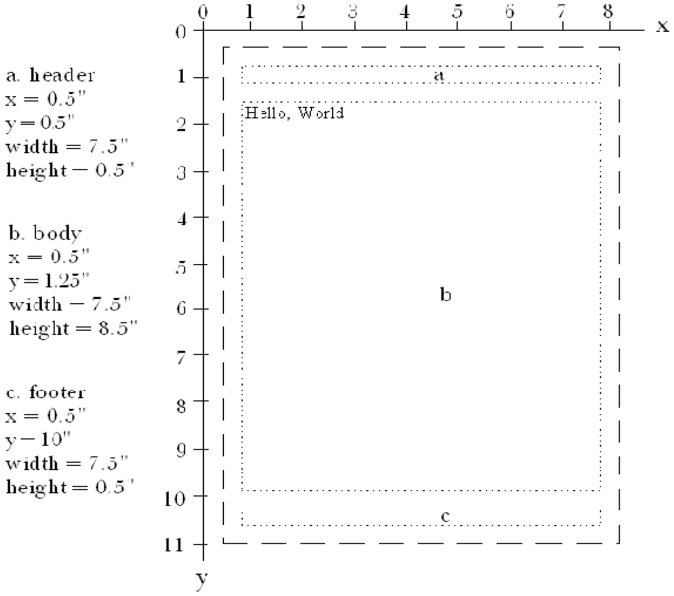


Figure 21 Header, Body, and Footer frames.

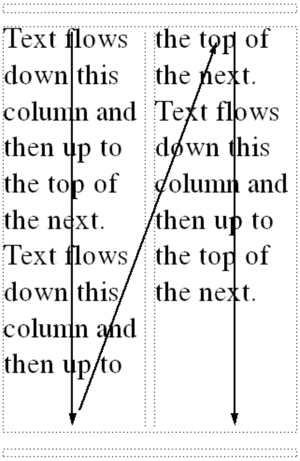
6.2 Multiple Columns

To create multiple columns, add `COLUMN COUNT` and `SPACING` attributes to the definition of the body frame in the XML template, for example:

```
<FRAME NAME="body">
<LOCATION X="0.5" Y="1.25"/>
<SIZE WIDTH="7.5" HEIGHT="8.5"/>
<COLUMN COUNT="2" SPACING="0.5"/>
```

The `COLUMN COUNT` and `SPACING` parameters instruct JClass PageLayout to flow text through the body frame in two columns, separated by a gap of 0.5”. Columns are always of equal width.

The changes to the template produce the following results.



Text flows down this column and then up to the top of the next. Text flows down this column and then up to the top of the next. Text flows down this column and then up to the top of the next.

Figure 22 Text flowing through columns.

6.3 Page Numbers

You number pages by embedding a macro in the frame that is to contain the page number, usually the header or the footer.

```
JCPage template_page = doc.stringToTemplate("bookLeft");
JCFrame footer_frame = template_page.stringToFrame(FOOTER);

JCTextStyle style = JCTextStyle.stringToStyle("default text");
try {
footer_frame.print(style, TextMacro.PAGE_NUMBER);
}
catch (EndOfFrameException e) {}
```

The preceding example uses the `stringToFrame` method to establish that the `footer_frame` is to hold the `PAGE_NUMBER` macro. Once the macro is embedded in the page template, it is added to the footer frame and evaluated whenever a new page is generated from this template.

Any content added to template pages should be done before any template pages are used, that is, before `JCFlow` is instantiated (otherwise content may not appear on the first page of the document). Please see Building Page Templates, in Chapter 2, for more information.

You must repeat this process for every page template used to generate document pages that contain page numbers.

```
template_page = doc.stringToTemplate("bookRight");
footer_frame = template_page.stringToFrame(FOOTER);

// print the page number macro to the frame
try {
footer_frame.print(style, TextMacro.PAGE_NUMBER);
}
catch (EndOfFrameException e) {}
```

In this manner, you have precise control over which pages display page numbers and which do not.

The TextMacro interface gives you the following options:

PAGE_NUMBER	Inserts the current page number.
ROMAN_NUMBER	Inserts the current page number in Roman numerals.
SECTION_NUMBER	Inserts the current section number.
SECTION_PAGE_NUMBER	Insets the current page number within the current section.
SECTION_PAGE_TOTAL	Inserts the total number of pages in the current section.
PAGE_TOTAL	Inserts the total number of pages in the document.

6.4 Creating Macros

In addition to the predefined macros in TextMacro (see Section 6.3, Page Numbers), JClass PageLayout allows you to create customized macros that allow the insertion of custom run-time text into a document.

To create your own macro, you must first create a java class that implements the TextMacro interface. (TextMacro is in the com.klg.jclass.page package.) TextMacro specifies three methods that must be implemented in your macro class: evaluate(), getStatus(), and getText(). Once the macro has been added to a document, each implemented method will be called by JClass PageLayout.

If getStatus() and getText() are called before evaluate(), you must ensure that they return MACRO_INITIALIZED and a non-null placeholder Sting, respectively. Once evaluate() is called by JClass PageLayout, the method should attempt to construct the text String that is represented by the macro.

If the macro can be evaluated given the current flow and page information, evaluate() must return MACRO_EVALUATED, as must any subsequent calls to getStatus(). Subsequent calls to getText() must return the evaluated text String.

If the macro cannot be evaluated given the current flow and page information, `evaluate()` must return `MACRO_NOT_YET_EVALUATED`, as must any subsequent calls to `getStatus()`. Subsequent calls to `getText()` must return a non-null placeholder String.

Note: `evaluate()` may be called by JClass PageLayout several times per macro (and may be passed different, potentially null values for the flow and page parameters). Once `evaluate()` returns `MACRO_EVALUATED` for the instance of the macro within the current frame, it is never called again. If the macro is being used in a static frame across multiple pages (for example, in the header of a table or a static page frame), the `evaluate()` method will be called again when any new frame containing the macro instance is created by JClass PageLayout.

For example, here is a class that overrides `TextMacro` and prints `continued` each time it is evaluated, except the first time.

```
import com.klg.jclass.page.*;

public class ContinuedMacro implements TextMacro {

    /** The text to which this macro has been evaluated */
    protected String text = "";

    /** The status of the result of the last evaluation */
    protected int status = TextMacro.MACRO_INITIALIZED;

    /** True the first time this macro is evaluated; false otherwise. */
    private boolean firstTime = true;

    public ContinuedMacro() {
    }

    /** Return currently evaluated text. */
    public String getText() {
        // if there is no current value for the macro, return
            placeholder text
        if (text == null) {
            return ("");
        }
        return (text);
    }

    /** Return current evaluation status. */
    public int getStatus() {
        return (status);
    }

    /** Evaluate macro. Parameters flow and page may be null. */
    public int evaluate(JCFlow flow, JCPage page) {
   // if flow or page is null, don't evaluate
        if (flow == null || page == null) {
            text = null;
            status = TextMacro.MACRO_NOT_YET_EVALUATED;
        } else {
            // first evaluation -- text is blank
            if (firstTime) {
                text = "";
                firstTime = false;
                status = TextMacro.MACRO_EVALUATED;
            // all other evaluations -- text is "continued"
            } else {
                text = "continued";
                status = TextMacro.MACRO_EVALUATED;
            }
        }
   // return evaluation status
        return status;
    }
}
```

Note: This macro can be used in many instances; for example, in table headers.

To add this macro to a frame, use:

```
try {
  frame.print(textStyle, new ContinuedMacro());
} catch (EndOfFrameException eofe) {
}
To add this macro to the flow, use:
flow.print(new ContinuedMacro());
```

6.5 Units of Measurement

Many JClass PageLayout functions require you to measure distances on the output page. For example, to import images or draw shapes, you must pinpoint the location on the page where the image or drawn object is to be placed. To do so, you must possess an understanding of the methods JClass PageLayout provides for the precise measurement of linear distances.

The `JCUnit` class lets you define linear distances using three different units of measurement: centimeters, inches, and points. You can set default units and convert distances from one unit type to the next. You can use `JCUnit.Point` to precisely identify a location on a page and `JCUnit.Margins` to create a margin on the inside of a frame.

6.5.1 Setting a Default Unit of Measurement

You can set the default unit type to be centimeters, inches, or points. Once you set a default unit, all methods that use measurement units use the default, unless instructed otherwise. For example, to set centimeters as the default unit type, enter:

```
JCUnit.setDefaultUnit(JCUnit.CM);
```

6.5.2 Converting Units of Measurement

Your application may need to convert a distance from one unit type to another on the fly. `JCUnit` provides methods to convert a distance to each supported unit type. For example, suppose you have defined a distance in centimeters, as follows:

```
JCUnit.Measure measurement = new JCUnit.Measure(JCUnit.CM, 5);
```

To convert `distance` to a measurement in inches, enter:

```
double distanceInInches = JCUnit.getAsInches(
    measurement.units, measurement.distance);
```

or more simply,

```
measurement.getAs(JCUnit.INCHES);
```

6.5.3 Defining Points

Some JClass PageLayout functions require you to define a location on a page, for example, in order to draw a line or a polygon. `JCUnit.Point` makes it possible for you to precisely define locations, using any of the available units of measurement.

```
    JCUnit.Point point = new JCUnit.Point(JCUnit.INCHES, 2.5, 2.5);
```

The preceding example pinpoints a location on the page at the X- and Y-coordinates of 2.5 inches by 2.5 inches. To draw a line or a polygon, you would define other points on the page and use the corresponding `JCFrame` method to connect those points with lines. For more information, refer to Section 6.9.1, Drawing Lines, and Section 6.9.5, Drawing Polygons.

6.5.4 Creating Margins

You can use `JCUnit.Margins` to set margins around the interior of a frame, for example, inside the body frame on your page.

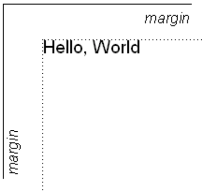


Figure 23   Margins inside a frame.

These margins are the same as those created when you define the frames in a page template. For more information, refer to Building Page Templates, in Chapter 2.

This example assumes that the `JCPage` template is named `template`.

```
    JCFrame frame = template.stringToFrame("body");
    frame.setMargins(new JCUnit.Margins((JCUnit.POINTS, 5, 5, 5, 5));
```

This example creates a margin of 5 points between the edge of the frame and any object rendered to it.

`JCUnit.Margins` provides methods that allow you to specify the width of individual margins.

JCUnit.Margins Method	Result
`setBottom()`	Sets the bottom margin to the width of the given `JCUnit.Measure`.

JCUnit.Margins Method	Result
setLeft()	Sets the left margin to the width of the given JCUnit.Measure.
setRight()	Sets the right margin to the width of the given JCUnit.Measure.
setTop()	Sets the top margin to the width of the given JCUnit.Measure.

6.6 Importing Images

To add an Image to a document, the first step is to load the image file. For example:

```
Image image = Toolkit.getDefaultToolkit().getImage("image.jpg");
```

This example instantiates the image as a java.awt.Image object and uses Toolkit.getDefaultToolkit().getImage() to load the image from its file source.

Next, you use a JCFrame or JCFlow embed(), float(), or paste() method to render the image into the current frame or flow. Recall that JCFrame methods render content apart from the main flow of the document, while their JCFlow counterparts render content into the flow.

Embedding places the image on the current line of text, which will wrap if there is not enough space on the current line to hold the image. For this reason, the embed method is often used for smaller graphics.

Floating an image inserts it on its own line. If there is not enough space on the page to hold the image, it "floats" to a roomier location, for example, the top of the next page. For this reason, the float method is often used for larger graphics.

Pasting an image locks the image at a set of coordinates you define. The paste() method is used to import images that must always appear at the same location, such as a logo in company letterhead. Pasting is only available as a JCFrame method.

In the following example, the image is resized to 50 by 50 points and embedded on the current line.

```
try {
flow.embedImage(image, new JCUnit.Dimension(JCUnit.POINTS,
50, 50));
}
catch (EndOfFrameException e) {
System.out.println(e.toString());
}
```

The following table describes the types of methods available for importing images.

JCFlow Method	Result
embedEPS()	Imports the image specified by EPSImage and places it on the current line of text.
embedIcon()	Imports the image specified by javax.swing.icon and places it on the current line of text.
embedImage()	Imports the image specified by java.awt.Image and places it on the current line of text.
floatEPS()	Imports the image specified by EPSImage and places it on its own line.
floatIcon()	Imports the image specified by javax.swing.icon and places it on its own line.
floatImage()	Imports the image specified by java.awt.Image and places it on its own line.
JCFrame.pasteEPS()	Imports the image specified by EPSImage and locks it to a specified location on the page.
JCFrame.pasteIcon()	Imports the image specified by javax.swing.icon and locks it to a specified location on the page.
JCFrame.pasteImage()	Imports the image specified by java.awt.Image and locks it to a specified location on the page.

6.6.1 Importing EPS Images

If your application prints to JCPostScriptPrinter, you can add EPS images to the document. You instantiate the file using EPSImage and provide an open reader to the EPS data.

```
EPSImage epsImage = new EPSImage(new BufferedReader(reader));
```

Next, you import the image using an embedEPS(), floatEPS(), or pasteEPS() method:

```
flow.floatEPS(epsImage, new JCUnit.Dimension(JCUnit.POINTS, 75, 100));
```

6.6.2 Importing Swing Icons

You can also import icons from the javax.swing.icon class into a JClass PageLayout document in much the same manner you import other images.

```
Icon icon = new ImageIcon(image);
flow.embedIcon(icon);
```

The preceding example creates a Swing icon based on the image defined earlier, and places it on the current line in the flow.

6.7 Displaying Imported Components

In addition to displaying images defined by image files, JClass PageLayout is capable of importing and displaying a visual object, such as a JClass Chart, so long as the object is of type `java.awt.Component`. The process is simple: instantiate the component and pass it to the current flow as a parameter in the flow's `embedComponent()` method.

Note: If you choose to embed a component using `embedComponent()`, the component cannot be changed until after the `JCDocument.print()` method has been executed. Because a document is only a series of information and references until it is printed, changing a component before it has been printed may change the output of the component, resulting in it being drawn differently than what was originally intended.

An example of a page containing a JClass Chart can be found in *ChartExample.java.* Once a component that knows how to draw itself is instantiated, one line is all that is needed to embed it in the flow:

```
// Create new chart instance.
chart = new JCChart();
// Load the chart's data from a data source
// so there is something to display, then embed it:
// ...
flow.embedComponent(chart); // Places the chart in the flow

// The following adds a caption to the embedded image:
flow.newLine();
flow.setCurrentTextStyle(JCTextStyle.ITALIC);
flow.print("Figure 1.1 A Simple Chart");
```

The image of the component may be drawn using:

```
flow.floatComponent(chart);
```

In this case, the image is drawn after the current line. If the command is encountered while there is a partial line being output, that line will be completed before the image is positioned on the page. The method takes alignment parameters to further control its position.

If you need to separate the image from the current line, bracket the call to `embedComponent()` with newlines, as follows,

```
flow.newLine();
flow.embedComponent(chart);
flow.newLine();
```

Method `embedComponent()` takes a `java.awt.Component` as a required parameter and two optional parameters: alignment and size. The alignment parameter takes one of the alignment constants in `JCDrawStyle` for positioning an object vertically relative to the line it is on. This parameter is useful for positioning a small component's image. The size parameter is specified with a `JCUnit.Dimension` object and permits scaling the image both horizontally and vertically.

Besides setting the size by passing a size parameter to `embedComponent()`, a component's printable size is determined:

- By the component's preferred size, if no other size setting has been made.
- By using the component's `setSize()` or `setPreferredSize()` method, if it has one.
- By its instantiated size if the application creates a instance of the component and displays it on-screen in a frame before flowing it on a page, as is done in *ChartExample.java*.

6.7.1 Native Scaling

You can use JClass PageLayout to print components and images to EPS, PS, PDF, or PCL. Note that you can embed components and images to achieve greater resolution through the use of the output type's native scaling. This process is for only PDF, PS and EPS (AWT printing is constantly at 72 dpi -- this is a Java limitation).

For instance, if you have an image that is 300 pixels by 300 pixels and you want to display it at a resolution of 300 dpi, you should embed the image at a size of 1 inch by 1 inch. Using this same image, if you wanted a resolution of 600 dpi, you would embed the image at a size of 0.5 inches by 0.5 inches.

6.8 Creating Draw Styles

Adding Borders, in Chapter 4 demonstrates the use of draw styles when drawing borders in a table. Later in Section 6.9, Drawing Shapes, you'll define a draw style to use.

Draw styles define the appearance of objects drawn on a page, such as the lines used in tables. `JCDrawStyle` provides methods and procedures for defining draw styles you can use to control the appearance of the lines and fills of the drawn objects in your document.

The following example defines a draw style named `ds`.

```
JCDrawStyle ds = JCDrawStyle.LINE;
```

6.8.1 Setting Line Properties

`JCDrawStyle` provides methods you can use to control the appearance of lines drawn in the draw style you've created. For example, to adjust the thickness of the line to 5 pts, enter:

```
ds.setLineWidth(new JCUnit.Measure(JCUnit.POINTS, 5));
```

To change the solid line to a dashed line, enter:

```
ds.setLineType(LINE_TYPE_BROKEN);
```

The following table describes the `JCDrawStyle` methods you can use to modify line styles.

JCDrawStyle Method	Result
`setDashLength`	Controls the length of the dashes and the spaces between them when line type is set to `LINE_TYPE_BROKEN`.
`setForegroundColor`	Controls the color of the line.
`setLineSpacing`	In a multi-line style, such as `LINE_TYPE_DOUBLE`, controls the amount of space left between the lines.
`setLineType`	Selects the appearance of the line. Options include: ■ `LINE_TYPE_BROKEN` ■ `LINE_TYPE_DOUBLE` ■ `LINE_TYPE_SINGLE`
`setLineWidth`	Uses `JCUnit.Measure` to control the width of the line.

6.8.2 Setting Fill Properties

By altering the draw style, you can adjust the fill color of two-dimensional objects, such as circles, rectangles, or polygons. `JCDrawStyle` provides separate methods for the specification of line and fill colors.

The following example modifies the draw style created in the previous section (`ds`), by setting its fill foreground color to yellow. You can specify any fill color you have defined using `java.awt.Color`.

```
ds.setFillForegroundColor(yellow);
```

6.9 Drawing Shapes

Using JClass PageLayout, your Java application can draw and print a variety of geometric shapes, including lines, circles, rectangles, rounded rectangles, and polygons.

To draw a shape, you must provide a `JCDrawStyle` that describes its appearance. (For more information, refer to Section 6.8, Creating Draw Styles.) In simple cases such as the following examples, you can use a default style, for example:

```
JCDrawStyle ds = JCDrawStyle.LINE;
```

6.9.1 Drawing Lines

To draw a line, you create an array of points, then call `JCFrame.drawLine()` to connect those points with a line drawn in the current `JCDrawStyle`.

```
ArrayList list = new ArrayList();
list.add(new JCUnit.Point(JCUnit.CM, 2, 3.5));
list.add(new JCUnit.Point(JCUnit.CM, 2, 5.5));
frame.drawLine(ds, list);
```

The preceding example draws a line between the two points defined in the array, as illustrated in the following diagram.

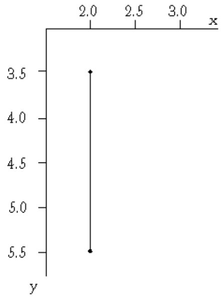


Figure 24   A drawn line.

6.9.2 Drawing Rectangles

To draw a rectangle, use `JCFrame.drawRectangle()`. For example:

```
frame.drawRectangle(ds, new JCUnit.Point(JCUnit.POINTS, 25, 100),
   new JCUnit.Dimension(JCUnit.POINTS, 50, 50));
```

The `drawRectangle()` method creates the outline of a rectangle. `JCUnit.Point` places the upper-left corner of the rectangle on the X- and Y-coordinates of 25 by 100 points. `JCUnit.Dimension` makes the rectangle a square by setting its size to 50 by 50 points.

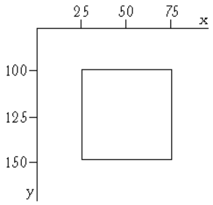


Figure 25   An outlined square (50 by 50).

To draw a filled rectangle, use `fillRectangle()`. The color used to fill the rectangle is the color defined by `JCDrawStyle.setFillForegroundColor()`.

```
frame.fillRectangle(ds, new JCUnit.Point(JCUnit.POINTS, 100, 100),
new JCUnit.Dimension(JCUnit.POINTS, 50, 50));
```

The sample code draws a square with a 100% black fill at the X- and Y-coordinates of 100 by 100 points, with dimensions of 50 by 50 points.

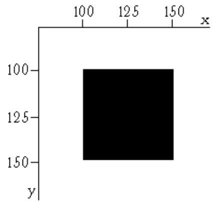


Figure 26 A filled square (50 by 50).

6.9.3 Drawing Rounded Rectangles

You can also draw outlined and filled rectangles with rounded corners. For rounded rectangles, you need to specify the radius of the rounded corners using `JCUnit.Measure`.

To draw the outline of a rounded rectangle:

```
frame.drawRoundedRectangle(ds, new JCUnit.Point(JCUnit.POINTS,
350, 350), new JCUnit.Dimension(JCUnit.POINTS, 50, 50),
new JCUnit.Measure (JCUnit.POINTS, 5));
```

The preceding example produces a rounded rectangle with dimensions of 50 by 50 points and a corner radius of 5 points:

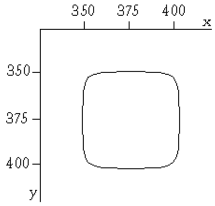


Figure 27 An outlined square with rounded corners.

To draw a filled, rounded rectangle:

```
frame.fillRoundedRectangle(ds, new JCUnit.Point(JCUnit.POINTS,
400, 400), new JCUnit.Dimension(JCUnit.POINTS, 50, 50),
new JCUnit.Measure (JCUnit.POINTS, 5));
```

The preceding example produces the following result:

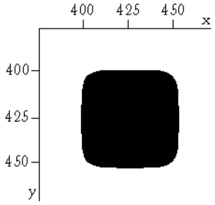


Figure 28 A filled square with rounded corners.

6.9.4 Drawing Circles

To draw a circle, use the `JCFrame.drawCircle()` method. For instance, here is the code to place the center of the circle at the X- and Y-coordinates of 175 by 175 points, and to set the radius of the circle to 25 points:

```
frame.drawCircle(ds, new JCUnit.Point(JCUnit.POINTS, 175, 175),
new JCUnit.Measure(JCUnit.POINTS, 25));
```

`JCUnit.Point` places the center of the circle at the X- and Y-coordinates of 175 by 175 points. `JCUnit.Measure` sets the radius of the circle to 25 points.

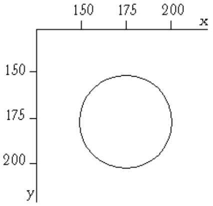


Figure 29 A circle with a radius of 25 points.

As with rectangles, you can draw circles filled with the color defined by the `JCDrawStyle`.

```
frame.fillCircle(ds, new JCUnit.Point(JCUnit.POINTS, 225, 225),
new JCUnit.Measure(JCUnit.POINTS, 25));
```

The preceding example produces the following result:

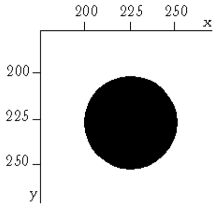


Figure 30   A filled circle with a radius of 25 points.

6.9.5 Drawing Polygons

JClass PageLayout also allows you to draw angular shapes with more than two sides – polygons. To draw a polygon, you create an `ArrayList` of points that define the X- and Y-coordinates of each of the polygon's corners. You then instruct JClass PageLayout to draw the polygon by connecting those points with lines.

For example, you could use the following code to draw a triangle:

```
ArrayList list = new ArrayList();
list.add(new JCUnit.Point(JCUnit.POINTS, 275, 250));
list.add(new JCUnit.Point(JCUnit.POINTS, 300, 300));
list.add(new JCUnit.Point(JCUnit.POINTS, 250, 300));
frame.drawPolygon(ds, list);
```

Using the X- and Y-coordinates as a guide, JClass PageLayout draws the following object:

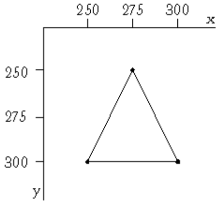


Figure 31   Constructing a polygon from a list of X- and Y-coordinates.

To draw more complex polygons, extend the list. For example, to draw a hexagon, define a total of six points in the list.

6.10 Render Objects

Render objects allow a description of the page-marking actions needed to draw a page stored in memory as the document is created. There are two kinds of render objects: those directly representing page-marking primitives (such as drawn lines or formatted text), and those representing additions to the flow of conceptual objects (such as horizontal rules). The current set of provided render objects includes all supported graphical primitives and all higher-level objects (for instance, images and tables) which can be added to the flow.

6.10.1 Render Object Categories

There are several interfaces that provide categorization of the types of render objects. The basic interfaces comprise:

- **Embedable**. Embedable objects can be added into the text flow as flowed objects because the interface provides the necessary support for determining the position of this object relative to the line contents. Text is not considered embedable because it defines the baseline of a line and doesn't need alignment attributes. `ImageRender` is the only Embedable render object.
- **Floatable**. Floatable objects can be added into the document flow as independent paragraph-level objects that do not break lines and allow normal flowed content to be added while they are awaiting sufficient space. `ImageRender` is the only Floatable object.
- **FlowMarker**. FlowMarker objects are not graphical primitives, but represent logical elements of the flow, which may have printer-specific or variable handling. The current FlowMarker objects are `HRuleMarker`, `ImageMarker`, and `TableMarker`.
- **Splitable**. Splitable objects are flow objects that implement methods that allow them to be broken into smaller pieces in order to fit on lines. The best example is text, which can be divided into words or even individual letters. `StringRender` is currently the only Splitable object.

6.10.2 Subclasses of the Render object

The `com.klg.jclass.page.render` package also contains classes that are not render objects; there are a number of page numbering and counting macros. The macro classes each implement a single page numbering or counting mechanism, such as representing the page number in roman numerals, and can be distinguished by names ending in `...Macro.java`. The following table describes all objects that can be considered render objects.

`ArcRender`	Represents a graphical primitive in the shape of a circular arc. A complete circle is a special case of this render object.
`BoxRender`	A graphical primitive describing a rectangular area.

HRuleMarker	Indicates that at this point in the flow, a line is drawn between the margins of the current frame. Storing a horizontal rule in this fashion allows the printer to make a decision about how it is reproduced rather than simply copying a stored line.
ImageMarker	Records the point in the flow at which a floating image was added. This marker allows the HTML printer to reposition images at their point of addition to the document, rather than at the point at which they may finally have been placed on a page.
ImageRender	Represents a drawable object such as an Image, an EPS file, or a Component.
LineRender	A graphical primitive for a line with an indefinite number of straight segments. Curves are not supported.
MacroRender	This render object stores a reference to a text macro that is to be evaluated and the result entered at the given point in the document.
RoundRectRender	Describes a rectangular graphical primitive with rounded corners.
StringRender	Represents all text elements in the document. As such it exports a number of methods designed to allow manipulation of the contents of text Strings.
SymbolRender	Instances of this class represent actions with respect to the text flow, such as newLine and newParagraph. A SymbolRender marking a new line action distinguishes an application-driven newLine() – the parent application called newLine() on the JCFlow or JCFrame – from an implicit newLine(), which occurs when a line becomes filled by the flowPrint() mechanism.
TableCellRender	This object locates and encapsulates a CellRenderer object so that the contents of a table cell can be drawn at the correct location.
TableMarker	Records the insertion in the document of a table, and subsequently the actual point of placement of a range of cells of the table.

6.11 Listening for JClass PageLayout Events

If your application needs to be informed about such events as the beginning, completion, or ending of a frame or a page, you can implement the JCFlowListener interface and examine the event to take appropriate action. The JCFlowListener implementation can either be passed to the JCFlow constructor or added to an existing JCFlow via the addFlowListener() method.

JCFlowEvent

A JCFlowEvent occurs when a flow enters or exits a new frame or page as a result of document flow, and also when a frame or page is marked as complete by the resolution of embedded macros. The methods in JCFlowEvent are:

JCFlowEvent Method	Description
getCurrentPageArea()	The current PageArea on which the event occurred.
getNextElementName()	The name of the next PageArea to be processed.
getNextPageArea()	The next page relative to where the event occurred.
getSource()	The source of the event, the current JCFlow where the event occurred.

JCFlowListener

JCFlowListener methods each take a JCFlowEvent as their only parameter.

JCFlowListener Method	Description
frameBegin()	Invoked before the flow to a frame begins.
frameComplete()	Invoked when the flow to a frame is complete, that is, when all macros in the frame have been evaluated.
frameEnd()	Invoked when the flow is transferred to another frame.
pageBegin()	Invoked before the flow to a page begins.
pageComplete()	Invoked when the flow to a page is complete, that is, when all macros on this page have been evaluated.
pageEnd()	Invoked when the flow is transferred to another page.

JCPrintEvent

A JCPrintEvent occurs when a document is opened or closed, or when a page begins or ends. The availability of a JCPrintEvent overcomes a limitation of AWT printing. You can now be notified when a document finishes printing.

JCPrintEvent Method	Description
getPageNumber()	Returns the page number or -1 if not applicable.

JCPrintEvent Method	Description
getEventId()	Returns the eventId. The returned value is one of JCPrintEvent.BEGIN_PAGE, JCPrintEvent.END_PAGE, JCPrintEvent.OPEN_DOCUMENT, or JCPrintEvent.CLOSE_DOCUMENT.

JCPrintListener

JCPrintListener methods each take a JCPrintEvent as their only parameter.

JCPrintListener Method	Description
openDocument()	Invoked before a document has been printed.
closeDocument()	Invoked after a document has been printed.
beginPage()	Invoked before a page has been printed.
endPage()	Invoked after a page has printed.

7

Printing Options

Printing to the System Printer ■ *Printing to a File*
Printing to a Screen ■ *Print Preview* ■ *OutputPolicy and FlushPolicy*

Creating a Printer, in Chapter 2, demonstrated that in order to print from a JClass PageLayout application you first need to set up a printer object that defines the nature of the print output. (We suggest that you buffer print output. See the note on Printing Large Documents, in Chapter 1, for details.) This chapter provides a more detailed explanation of the print options JClass PageLayout makes available to you.

7.1 Introduction

The printer associated with a document is used to determine text size in order to perform text layout. Since the printer initially associated with a document is used to size and lay out the text, using a different printer to create the output stream or image may result in spacing problems.

Please note that a document may hang when it is printing. This can happen if the flow contains a floating object that is too large to fit in any frame in the document. The flow will attempt to past all floating objects as part of completing the document before printing it, which includes pasting any floating objects that haven't yet been fit into the document. If the floating object does not fit into the current frame, the flow will traverse to the next frame, possibly generating a new page, and try there. If there is no frame in the document large enough to contain the object, then a simple infinite loop will result.

7.2 Printing to the System Printer

JClass PageLayout provides two different printing formats that send output to your system printer. `JCAWTPrinter` allows you to render most AWT components in accordance with the Java 2 Printing API.

7.2.1 Using JCAWTPrinter

Printing to the system printer using JCAWTPrinter is much slower than using other JClass PageLayout printing methods. JCAWTPrinter uses the Java 2 Printing API, and supports most of the Graphics 2D API. The Java 2 Printing API usually causes all Java 2D graphics to be rendered on the client machine before sending a raster image to the printer. When this raster printing method is used, the amount of data sent to the printer is much greater than the amount of data created by other printers. This aspect of the Java 2 Printing API is the reason why using JCAWTPrinter to print to the system printer is not as efficient as using the other printing methods supported by JClass PageLayout.

There is an optimized shape printing method that can be used in place of the raster printing method above under certain conditions. If there are no images on the page, and if only solid colors are used (java.awt.Color), then the shape printing method will be used. This optimized process converts the graphics on the page into shapes, which are then filled. Less data is generated using this process instead of the raster printing method, although a large amount of data can still be generated for complex shapes (such as a page of text). The shape printing method case is being extended, so printing performance should be better in the next Java 2 platform release.

Java requires that you instantiate JCAWTPrinter in a try block. To obtain the best results, the page templates list, which will later be passed to the JCDocument constructor, should also be passed to the JCAWTPrinter constructor. It is possible to use the current printer driver and not pop up the print dialog. There is a constructor for a JCAWTPrinter which takes a boolean that determines whether to pop up the dialog, as shown in the following code fragment:

```
JCPrinter printer = null;
try {
     // Create an AWT printer, output to printer. Don't show print dialog
     printer = new JCAWTPrinter(false, templates);
}
catch (JCAWTPrinter.PrinterJobCancelledException e) {
     System.out.println("Print Job Cancelled by user");
     System.exit(1);
}
```

After programming the output, send it to the system printer.

```
document.print();
```

7.3 Printing to a File

You may already know that java.io.FileOutputStream lets you write data to a file. JCHTMLPrinter, JCPCLPrinter, JCPDFPrinter, and JCPostScriptPrinter format file output to enable your application to print directly to HTML, PCL, PDF, and PostScript files.

7.3.1 Printing to a PostScript File

The following example generates an Adobe PostScript file using the JCPostScriptPrinter.

1. Create an instance of the FileOutputStream class.

   ```
   try {
   outfile = new FileOutputStream("test.ps");
   }
   catch (FileNotFoundException e) {
   System.out.println("Could not open file");
   return;
   }
   ```

2. Next, instantiate the printer object, selecting outfile as its output stream.

   ```
   printer = new JCPostScriptPrinter(outfile);
   ```

3. To generate the file, call the JCDocument.print() method.

7.3.2 Printing to a PDF File

The following example generates an Adobe PDF file using the JCPDFPrinter.

1. Create an instance of the FileOutputStream class.

   ```
   try {
   outfile = new FileOutputStream("test.pdf");
   }
   catch (FileNotFoundException e) {
   System.out.println("Could not open file");
   return;
   }
   ```

2. Instantiate the printer object, selecting outfile as the output stream.

   ```
   printer = new JCPDFPrinter(outfile);
   ```

3. To generate the file, call the JCDocument.print() method.

7.3.3 Printing to a PCL file

The following example generates a Hewlett-Packard PCL file using the JCPCLPrinter.

1. Create an instance of the FileOutputStream class.

   ```
   try {
   outfile = new FileOutputStream("test.pcl");
   }
   catch (FileNotFoundException e) {
   System.out.println("Could not open file");
   return;
   }
   ```

2. Instantiate the printer object, selecting outfile as the output stream.

   ```
   printer = new JCPCLPrinter(outfile);
   ```

3. To generate the file, call the JCDocument.print() method.

7.3.4 Printing to an HTML file

The following example generates an HTML file using the JCHTMLPrinter.

1. Create an instance of the `FileOutputStream` class.
   ```
   try {
   outfile = new FileOutputStream("test.html");
   }
   catch (FileNotFoundException e) {
   System.out.println("Could not open file");
   return;
   }
   ```
2. Instantiate the printer object, selecting `outfile` as the output stream.
   ```
   printer = new JCHTMLPrinter(outfile);
   ```
3. To generate the file, call the `JCDocument.print()` method.

Because the HTML specification lacks many of the features normally associated with printed documents you will encounter certain limitations if you decide to create HTML pages from JClass PageLayout. Among the limitations are:

- No pagination – HTML has no tag for a page, so the document becomes a single HTML file. Users scroll through the file using their browser. Whatever pagination information the source file contains is ignored.
- Components/images – HTML pages created with JClass PageLayout cannot include components or images.
- Tables – HTML does define a table tag, but the mechanism is very different from that used by most page layout applications. Not all of your table formats are available when exporting to HTML.
- Single header and footer – Because the concept of a page is lost in HTML, different page headers and footers throughout the document are ignored. Instead, JClass PageLayout attempts to find a header and a footer to be used as header and footer for the entire HTML document.

7.4 Printing to a Screen

You can use `JCAWTScreenPrinter` to create a Java `JComponent` on which it will render your print output. You can then embed the component in your application. For example, in JClass PageLayout, we use `JCAWTScreenPrinter` to create the print previewer described in the next section.

1. First, get the component from the printer:
   ```
   JCPrintPage page = (JCPrintPage)
   ((JCAWTScreenPrinter)printer).getComponent();
   ```

   The returned component is a subclass of `JComponent` called `JCPrintPage`.
2. Set the document and page you want to render to the component:
   ```
   page.setDocument(document);
   page.setPage(0);
   ```

3. Add the frame to the component and make it visible:

   ```
   frame.getContentPane().add(page);
   frame.pack();
   frame.setVisible(true);
   ```

7.5 Print Preview

End users have grown accustomed to the "Print Preview" – an interim step that displays documents as they will appear when printed. Print previews make it possible for the user to page through their documents and scan for layout errors before they send the document to the printer. JClass PageLayout brings this functionality to Java applications using the JCAWTPreviewer class.

As described in the previous section, JCAWTPreviewer requires java.awt.Component and JCAWTScreenPrinter to realize the component on which the frame will be drawn. The next step is to use a JCAWTPreviewer constructor to declare the title that will be displayed in the frame, the name of the frame declared in your program, and the name of the document you want to display in the frame.

```
JCAWTPreviewer previewer = new JCAWTPreviewer("Print Preview",
frame, document);
previewer.setVisible(true);
```

As well, in order for the JCAWTPreviewer to work correctly, the FlushPolicy must be set to FLUSH_POLICY_ALWAYS_SAVE. (For details on FlushPolicy, please refer to FlushPolicy.)

In print preview mode, the *Hello, World* program you saw in JClass PageLayout Basics, in Chapter 1, might appear as follows:

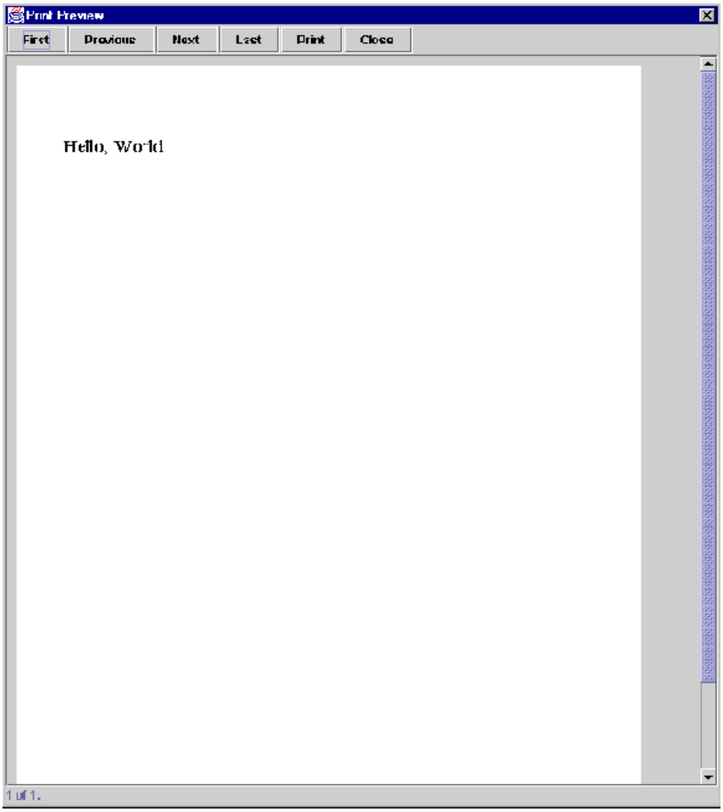


Figure 32 Print preview.

7.6 OutputPolicy and FlushPolicy

Once a page is created, you can designate whether it will be held until the entire document is finished before it is printed. This is set via the `outputPolicy` property; please see OutputPolicy.

As well, once pages are printed, you can designate whether pages are to be discarded once printed. You can do this by setting the `flushPolicy` property. Please see FlushPolicy.

OutputPolicy

Once a page is created, you can designate whether it will be held until the entire document is finished before it is printed. In the JCDocument class, the outputPolicy property indicates whether rendered pages are to be held for printing. If set to OUTPUT_POLICY_ON_REQUEST (default), then completed pages are held in memory until the document is printed. If set to OUTPUT_POLICY_IMMEDIATE, then each page is outputted as it is completed (if all predecessors are complete).

The getOutputPolicy() method gets the document's current policy on outputting completed (rendered) pages. The setOutputPolicy() method sets the document's behavior for outputting printed pages. Its parameter, outputPolicy, indicates the policy to apply to completed (rendered) pages.

FlushPolicy

Once pages are printed, you can designate whether to save or flush them. In the JCDocument class, the flushPolicy property indicates whether pages are to be discarded once printed. If set to FLUSH_POLICY_ALWAYS_SAVE (default), then all pages are saved, not flushed, as they are completed and printed. If set to FLUSH_POLICY_ON_OUTPUT, then pages are flushed as they are completed and printed.

The getFlushPolicy() method gets the document's current policy on flushing completed (printed) pages. The setFlushPolicy() method sets the document's behavior for flushing printed pages. The flushPolicy parameter of setFlushPolicy() designates the policy to apply to completed (output) pages.

In order for the JCAWTPreviewer to work correctly, the FlushPolicy must be set to FLUSH_POLICY_ALWAYS_SAVE.

Part II

Reference Appendices

Appendix A

JClass PageLayout Design Elements

Page Templates ■ *Controlling Flow* ■ *Standard Styles* ■ *Alignment* ■ *Indents*
Tab Alignment ■ *Table Style* ■ *Line Style* ■ *Cell Alignment*

This appendix summarizes JClass PageLayout design elements.

A.1 Page Templates

Default Page Templates

Default Template	Description
`Blank_A3`	Creates a blank (no headers or footers) page of standard ISO A3 size.
`Blank_A4`	Creates a blank (no headers or footers) page of standard ISO A4 size.
`Blank_A5`	Creates a blank (no headers or footers) page of standard ISO A5 size.
`Blank_8p5x11`	Creates a blank (no headers or footers) page of standard US Letter size.
`Blank_8p5x14`	Creates a blank (no headers or footers) page of standard US Legal size.
`Blank_11x17`	Creates a blank (no headers or footers) page of standard Tabloid size.

Template Elements and Attributes

Element	Attributes	Child Elements
<JCPAGETEMPLATE>	TITLE: An optional attribute that names this page template.	<PAGE>
<PAGE>	NAME: Required. The name of this page type, referenced by other page definitions using the <FLOWPAGE> tag. UNIT: The unit of measurement used to plot out this page. Choose from inches, points, cm, cms, centimetres, and centimeters. The default is inches. COLOR: Optional. Specifies a default background color for this page using hexadecimal notation or a color from com.klg.jclass.util.swing. JCSwingTypeConverter. ORIENTATION: Choose from automatic, portrait, and landscape. FIRST: A Boolean attribute that indicates whether or not this page template is used for the first page in the document. The default is false.	<LOCATION>, <SIZE>, <FRAME>+, <FLOWFRAME>*, <FLOWPAGE>, <FLOWSECTION>
<FRAME>	NAME: Required. The name of this frame type, referenced by other page definitions using the <FLOWFRAME> tag. UNIT: The unit of measurement used to plot out this frame. Choose from inches, points, cm, cms, centimetres, and centimeters. The default is inches. COLOR: Optional. Specifies a default background color for this frame using hexadecimal notation or a color from com.klg.jclass.util.swing. JCSwingTypeConverter.	<LOCATION>, <SIZE>, <BORDER>?, <COLUMN>?
<LOCATION>	X: Required. Specifies the distance of the page or frame from the left-hand page edge. Y: Required. Specifies the distance of the page or frame from the top of the page.	None.

$^{+}$ Required and repeatable.
* Optional and repeatable.
$^{?}$ Optional and non-repeatable.

Element	Attributes	Child Elements
`<SIZE>`	`WIDTH`: Required. Specifies the width of the page or frame, measured in the units defined by the `<UNIT>` tag. `HEIGHT`: Required. Specifies the height of the page or frame, measured in the units defined by the `<UNIT>` tag.	None.
`<BORDER>`	`TYPE`: Specifies the style used to draw a frame border. Choose from `blank`, `broken`, `dashed`, `double`, `none`, `plain`, `regular`, or `single`. The default is `blank`. `COLOR`: Specifies the border color using either hexadecimal notation or a color from `java.awt.Color`. The default is `black`. `THICKNESS`: Optional. Specifies the border width in pixels. The default is `0.1`.	None.
`<COLUMN>`	`COUNT`: Required. Specifies the number of columns in the frame. `SPACING`: Optional. Specifies the amount of space left between columns, measured in the units defined by the `<UNIT>` tag.	None.
`<FLOWFRAME>`	`NAME`: Required. Specifies the name of a frame to be added to the sequence of frames to which the document will flow content.	None.
`<FLOWPAGE>`	`NAME`: Required. Specifies the name of the page to which the flow is to progress when a new page is begun.	None.
`<FLOWSECTION>`	`NAME`: Required. Specifies the name of the page to which the flow is to progress when a new section is begun.	None.

+ Required and repeatable.
* Optional and repeatable.
? Optional and non-repeatable.

A.2 Controlling Flow

Frame Method

JCFrame Method	Description
`newColumn()`	Generates a column break and advances the text to the next column in the specified frame. Throws an `EndOfFrameException` in the last (only) column of a frame.
`newLine()`	Ends the current line and transfers the flow to a new line. Throws an `EndOfFrameException` if there is not enough room to print the text.
`print()`	Renders the specified content to this frame. Throws an `EndOfFrameException` if there is not enough room to print the text.

Flow Method

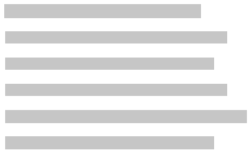
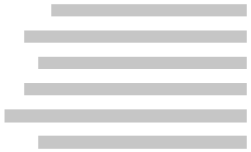
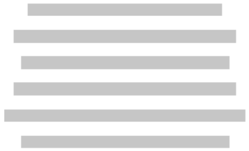
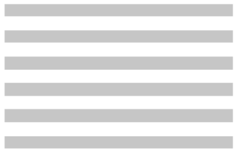
JCFlow Method	Description
`print()`	Renders the specified content to the flow.
`newLine()`	Ends the current line and begins a new line.
`newParagraph()`	Begins a new paragraph.
`newColumn()`	Advances the text flow to the top of the next column, in the next frame if necessary.
`newFrame()`	Advances the text flow to the next frame, generating a new page if necessary.
`newPage()`	Creates a new page based on the current page's `FlowPageTemplate`, and directs the flow to the first frame of the new page's `FlowFrameList`.
`newSection()`	Creates a new section based on the current section's `FlowSectionTemplate`, and directs the flow to the first frame of the new page's `FlowFrameList`.

A.3 Standard Styles

Style	Appearance
`BOLD`	Left-aligned, single-spaced, 10 pt. bold Times New Roman.

Style	Appearance
BOLD_ITALIC	Left-aligned, single-spaced, 10 pt. bold, italic Times New Roman.
CODE	Left-aligned, single-spaced, 10 pt. plain Courier.
CODE_INDENTED	Left-aligned, single-spaced, 10 pt. plain Courier with left, right, and paragraph indents of 0.25”.
DEFAULT_HEADER	Center-aligned, single-spaced, 14 pt. bold Times New Roman.
DEFAULT_TEXT	Left-aligned, single-spaced, 12 pt. plain Times New Roman.
HEADING	Left-aligned, single-spaced, 10 pt. plain Helvetica.
HEADING_BOLD	Left-aligned, single-spaced, 10 pt. bold Helvetica.
HEADING1	Left-aligned, single-spaced, 18 pt. bold Helvetica.
HEADING2	Left-aligned, single-spaced, 18 pt. plain Helvetica.
HEADING3	Left-aligned, single-spaced, 16 pt. bold Helvetica.
HEADING4	Left-aligned, single-spaced, 16 pt. plain Helvetica.
HEADING5	Left-aligned, single-spaced, 14 pt. bold Helvetica.
HEADING6	Left-aligned, single-spaced, 14 pt. plain Helvetica.
HEADING7	Left-aligned, single-spaced, 12 pt. bold Helvetica.
INDENTED	Left-aligned, single-spaced, 10 pt. plain Times New Roman, with left, right, and paragraph indents of 0.25”.
ITALIC	Left-aligned, single-spaced, 10 pt. italic Times New Roman.
NORMAL	Left-aligned, single-spaced, 10 pt. plain Times New Roman.
PLAIN	Left-aligned, single-spaced, 10 pt. plain Times New Roman.

A.4 Alignment

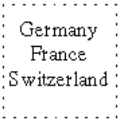
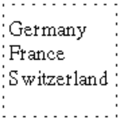
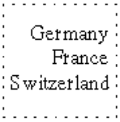
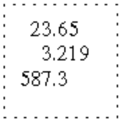
Parameter	Result	
`ALIGNMENT_LEFT`	Paragraph text is left aligned.	
`ALIGNMENT_RIGHT`	Paragraph text is right aligned.	
`ALIGNMENT_CENTER`	Paragraph text is center aligned.	
`ALIGNMENT_JUSTIFY`	Paragraph text is left and right aligned.	

A.5 Indents

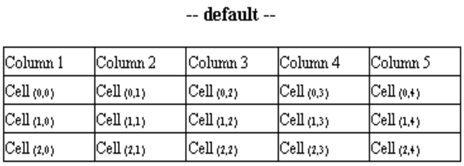
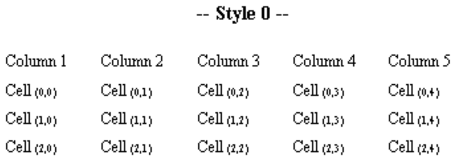
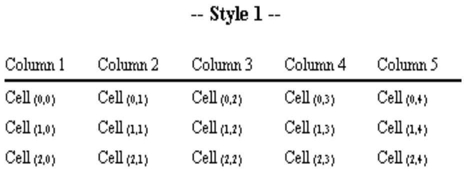
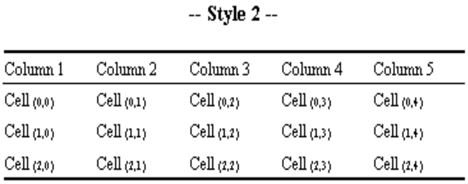
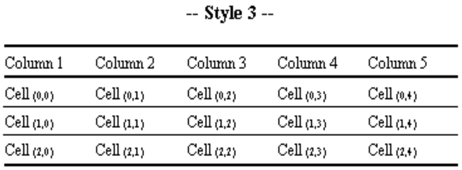
JCTextStyle Method	Description
`setLeftIndent()`	Defines the amount of space between the left side of the frame and the left edge of every line in the paragraph except for the first line.

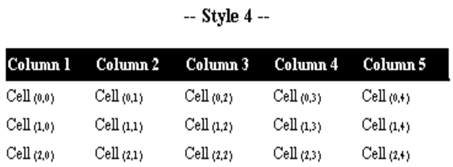
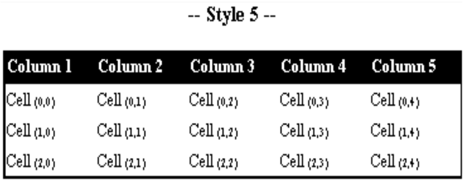
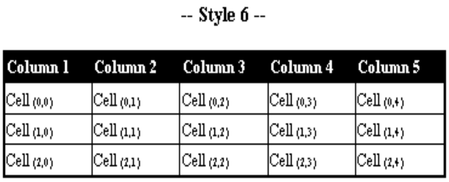
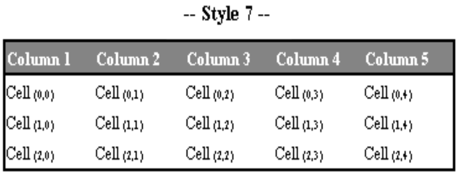
JCTextStyle Method	Description
setParagraphIndent()	Defines the amount of space between the left side of the frame and the left edge of the first line in the paragraph.
setRightIndent()	Defines the amount of space between the right side of the frame and the right edge of the paragraph.

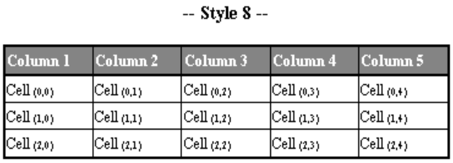
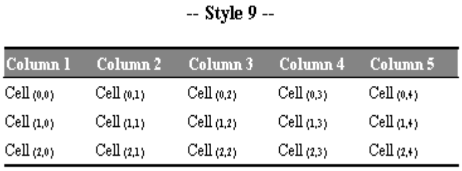
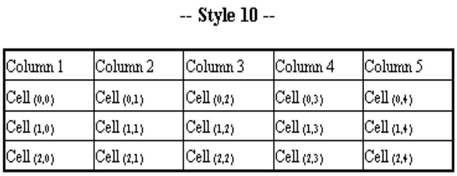
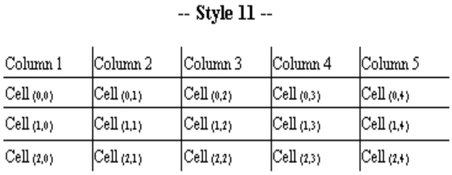
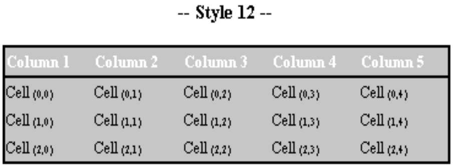
A.6 Tab Alignment

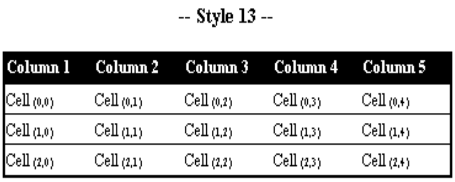
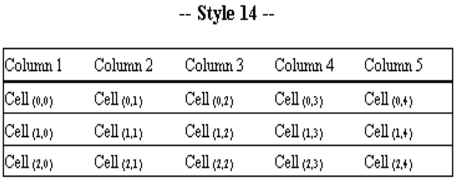
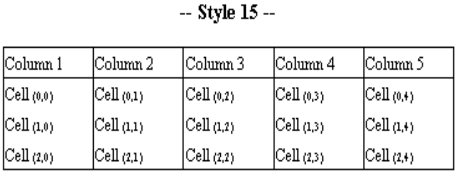
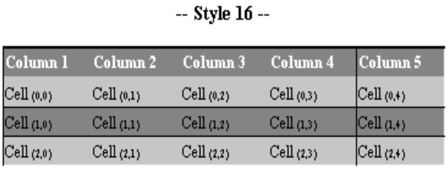
Field	Result	
TAB_ALIGNMENT_CENTER	Aligns an equal amount of text on either side of the tab stop.	 Germany France Switzerland
TAB_ALIGNMENT_LEFT	Aligns the left side (beginning) of the text with the tab stop.	 Germany France Switzerland
TAB_ALIGNMENT_RIGHT	Aligns the right side (end) of the text with the tab stop.	 Germany France Switzerland
TAB_ALIGNMENT_DECIMAL	Aligns a decimal or period (.) in the text with the tab stop. Primarily used for aligning columns of numbers.	 23.65 3.219 587.3

A.7 Table Style

Style Name	Description	Image
`Default`	■ thin left, right, top, bottom, horizontal, and column borders ■ no column shading ■ regular heading font	 -- default -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style0`	■ no left, right, top, bottom, horizontal, or column borders ■ no column shading ■ regular heading font	 -- Style 0 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 1`	■ thin header border; no other borders ■ no column shading ■ regular heading font	 -- Style 1 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 2`	■ thin header border; thick top and bottom borders; no other borders ■ no column shading ■ regular heading font	 -- Style 2 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 3`	■ thick top, bottom, and header borders; thin horizontal borders; no column border ■ no column shading ■ regular heading font	 -- Style 3 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)

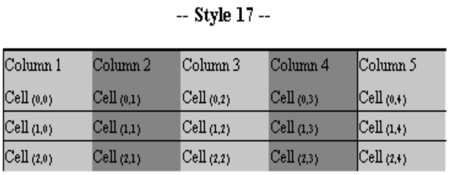
Style Name	Description	Image
`Style 4`	- no left, right, top, bottom, horizontal, or column borders - header colored (black); reverse type for header text - no column shading	-- Style 4 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 5`	- thick right, left, top, bottom, and header borders; no column or horizontal borders - header colored (black); reverse type for header text - no column shading	-- Style 5 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 6`	- thick right, left, top, bottom and header borders; thin horizontal and column borders - header colored (black); reverse type for header text - no column shading	-- Style 6 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 7`	- thick right, left, top, bottom and header borders; no horizontal or column borders - header colored (gray); reverse type for header text - no column shading	-- Style 7 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)

Style Name	Description	Image
`Style 8`	■ thick right, left, top, bottom, and header borders; thin horizontal and column borders ■ header colored (gray); reverse type for header text ■ no column shading	 -- Style 8 -- Column 1 · Column 2 · Column 3 · Column 4 · Column 5 Cell (0,0) · Cell (0,1) · Cell (0,2) · Cell (0,3) · Cell (0,4) Cell (1,0) · Cell (1,1) · Cell (1,2) · Cell (1,3) · Cell (1,4) Cell (2,0) · Cell (2,1) · Cell (2,2) · Cell (2,3) · Cell (2,4)
`Style 9`	■ thin header border; thick top, and bottom borders; no right, left, horizontal, and column borders ■ header colored (gray); reverse type for header text ■ no column shading	 -- Style 9 -- Column 1 · Column 2 · Column 3 · Column 4 · Column 5 Cell (0,0) · Cell (0,1) · Cell (0,2) · Cell (0,3) · Cell (0,4) Cell (1,0) · Cell (1,1) · Cell (1,2) · Cell (1,3) · Cell (1,4) Cell (2,0) · Cell (2,1) · Cell (2,2) · Cell (2,3) · Cell (2,4)
`Style 10`	■ thick right, left, top, and bottom borders; no header, horizontal, and column borders ■ regular heading font ■ no column shading	 -- Style 10 -- Column 1 · Column 2 · Column 3 · Column 4 · Column 5 Cell (0,0) · Cell (0,1) · Cell (0,2) · Cell (0,3) · Cell (0,4) Cell (1,0) · Cell (1,1) · Cell (1,2) · Cell (1,3) · Cell (1,4) Cell (2,0) · Cell (2,1) · Cell (2,2) · Cell (2,3) · Cell (2,4)
`Style 11`	■ no right, left, top, and bottom borders; thin header, horizontal, and column borders ■ plain headers ■ no column shading	 -- Style 11 -- Column 1 · Column 2 · Column 3 · Column 4 · Column 5 Cell (0,0) · Cell (0,1) · Cell (0,2) · Cell (0,3) · Cell (0,4) Cell (1,0) · Cell (1,1) · Cell (1,2) · Cell (1,3) · Cell (1,4) Cell (2,0) · Cell (2,1) · Cell (2,2) · Cell (2,3) · Cell (2,4)
`Style 12`	■ thick right, left, top, bottom, and header borders; no horizontal or column borders ■ header colored (gray); reverse type for header text ■ gray column shading	 -- Style 12 -- Column 1 · Column 2 · Column 3 · Column 4 · Column 5 Cell (0,0) · Cell (0,1) · Cell (0,2) · Cell (0,3) · Cell (0,4) Cell (1,0) · Cell (1,1) · Cell (1,2) · Cell (1,3) · Cell (1,4) Cell (2,0) · Cell (2,1) · Cell (2,2) · Cell (2,3) · Cell (2,4)

Style Name	Description	Image
`Style 13`	■ thick right, left, top, bottom, and header borders; thin horizontal border; no column borders ■ header colored (black); reverse type for header text ■ no column shading	 -- Style 13 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 14`	■ thin right, left, top, bottom, and horizontal borders; thick header border; no column border ■ plain header ■ no column shading	 -- Style 14 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 15`	■ thin right, left, top, bottom, header, and column borders; no horizontal border ■ plain header ■ no column shading	 -- Style 15 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)
`Style 16`	■ thin left and horizontal borders; thick top border; no bottom, right, and header borders; thin border between last two columns ■ header colored (dark gray); reverse type for header text ■ no column shading ■ alternate row shading (light gray and dark gray)	 -- Style 16 -- Column 1 Column 2 Column 3 Column 4 Column 5 Cell (0,0) Cell (0,1) Cell (0,2) Cell (0,3) Cell (0,4) Cell (1,0) Cell (1,1) Cell (1,2) Cell (1,3) Cell (1,4) Cell (2,0) Cell (2,1) Cell (2,2) Cell (2,3) Cell (2,4)

<table>
<tr><th>Style Name</th><th>Description</th><th>Image</th></tr>
<tr><td>Style 17</td><td>

- thin left and horizontal borders; thick top border; no bottom, right and header borders; thin border between last two columns
- header reverse type
- alternate row shading (light gray and dark gray)

</td><td>


-- Style 17 --

<table>
<tr><th>Column 1</th><th>Column 2</th><th>Column 3</th><th>Column 4</th><th>Column 5</th></tr>
<tr><td>Cell (0,0)</td><td>Cell (0,1)</td><td>Cell (0,2)</td><td>Cell (0,3)</td><td>Cell (0,4)</td></tr>
<tr><td>Cell (1,0)</td><td>Cell (1,1)</td><td>Cell (1,2)</td><td>Cell (1,3)</td><td>Cell (1,4)</td></tr>
<tr><td>Cell (2,0)</td><td>Cell (2,1)</td><td>Cell (2,2)</td><td>Cell (2,3)</td><td>Cell (2,4)</td></tr>
</table>
</td></tr>
</table>

A.8 Line Style

LINE_TYPE_BROKEN	Applies a dotted line to the table border.
LINE_TYPE_DOUBLE	Applies a double line to the table border.
LINE_TYPE_NONE	Blanks out the table border.
LINE_TYPE_SINGLE	Applies a single line to the table border.

A.9 Cell Alignment

CELL_ALIGNMENT_CENTER	Aligns cell text to the middle of the row.
CELL_ALIGNMENT_TOP	Aligns cell text to the top of the row.
CELL_ALIGNMENT_BOTTOM	Aligns cell text to the bottom of the row.
CELL_ALIGNMENT_NONE	No specific cell alignment.

Appendix B

JClass PageLayout Commonly Used Methods

This appendix summarizes JClass PageLayout commonly used methods in alphabetical order.

B.1 JCDrawStyle

Method	Description
`setDashLength`	Controls the length of the dashes and the spaces between them when line type is set to `LINE_TYPE_BROKEN`.
`setForegroundColor`	Controls the color of the line.
`setLineSpacing`	In a multi-line style, such as `LINE_TYPE_DOUBLE`, controls the amount of space left between the lines.

`setLineType`	Selects the appearance of the line. Options include:
	`LINE_TYPE_BROKEN`
	`LINE_TYPE_DOUBLE`
	`LINE_TYPE_SINGLE`

`setLineWidth`	Uses `JCUnit.Measure` to control the width of the line.

B.2 JCFlow

Method	Description
`embedEPS()`	Imports the image specified by `EPSImage` and places it on the current line of text.

Method	Description
embedIcon()	Imports the image specified by javax.swing.icon and places it on the current line of text.
embedImage()	Imports the image specified by java.awt.Image and places it on the current line of text.
floatEPS()	Imports the image specified by EPSImage and places it on its own line.
floatIcon()	Imports the image specified by javax.swing.icon and places it on its own line.
floatImage()	Imports the image specified by java.awt.Image and places it on its own line.
JCFrame.pasteEPS()	Imports the image specified by EPSImage and locks it to a specified location on the page.
JCFrame.pasteIcon()	Imports the image specified by javax.swing.icon and locks it to a specified location on the page.
JCFrame.pasteImage()	Imports the image specified by java.awt.Image and locks it to a specified location on the page.
newColumn()	Advances the text flow to the top of the next column, in the next frame if necessary.
newFrame()	Advances the text flow to the next frame, generating a new page if necessary.
newLine()	Ends the current line and begins a new line.
newPage()	Creates a new page based on the current page's FlowPageTemplate, and directs the flow to the first frame of the new page's FlowFrameList.
newParagraph()	Begins a new paragraph.
newSection()	Creates a new section based on the current section's FlowSectionTemplate, and directs the flow to the first frame of the new page's FlowFrameList.
print()	Renders the specified content to the flow.

B.3 JCFlowEvents

Method	Description
getCurrentPageArea()	The current PageArea on which the event occurred.
getNextElementName()	The name of the next PageArea to be processed.
getNextPageArea()	The next page relative to where the event occurred.
getSource()	The source of the event, the current JCFlow where the event occurred.

B.4 JCFlowListener

Method	Description
frameBegin()	Invoked before the flow to a frame begins.
frameComplete()	Invoked when the flow to a frame is complete, that is, when all macros in the frame have been evaluated.
frameEnd()	Invoked when the flow is transferred to another frame.
pageBegin()	Invoked before the flow to a page begins.
pageComplete()	Invoked when the flow to a page is complete, that is, when all macros on this page have been evaluated.
pageEnd()	Invoked when the flow is transferred to another page.

B.5 JCFrame

Method	Description
newColumn()	Generates a column break and advances the text to the next column in the specified frame. Throws an EndOfFrameException in the last (only) column of a frame.
newLine()	Ends the current line and transfers the flow to a new line. Throws an EndOfFrameException if there is not enough room to print the text.
print()	Renders the specified content to this frame. Throws an EndOfFrameException if there is not enough room to print the text.

B.6 JCPageTable

Method	Description
`setAllBorders()`	Applies the style to every border in the table.
`setBottomBorder()`	Applies the style to the bottom border of the table's last row.
`setColumnBorder()`	Applies the style to all column borders, except for the perimeter borders Left and Right.
`setEdgeBorders()`	Applies the style to all borders on the perimeter of the table (Top, Bottom, Left, and Right).
`setHeaderBorder()`	Applies the style to the border between the header table and the table's first row.
`setHorizontalBorder()`	Applies the style to the Top, Bottom, and header borders.
`setInternalBorders()`	Applies the style to all row and column borders except for the perimeter borders.
`setLeftBorder()`	Applies the style to the border on the table's left-hand side.
`setRightBorder()`	Applies the style to the border on the table's right-hand side.
`setRowBorder()`	Applies the style to all table row borders except for the header border and the perimeter borders Top and Bottom.
`setTopBorder()`	Applies the style to the top border of the table's first row. If there is a header table, applies the style to the top row of the header.
`setVerticalBorders()`	Applies the style to column, Left, and Right borders.

B.7 JCPageTemplate

Method	Description
`importTemplates(JCDocument doc, File xmlfile)`	Reads from `java.io.File` to import an XML template and apply it to the specified `JCDocument`.
`importTemplates(JCDocument doc, InputSource input)`	Reads from `org.xml.sax.InputSource` to import an XML template and apply it to the specified `JCDocument.`

Method	Description
importTemplates(JCDocument doc, Reader reader)	Reads from java.io.Reader to import an XML template and apply it to the specified JCDocument.
loadTemplates(File xmlfile)	Reads from java.io.File to load the XML template without applying it to a specific document.
loadTemplates(InputSource input)	Reads from org.xml.sax.InputSource to load the XML template without applying it to a specific document.
loadTemplates(Reader reader)	Reads from java.io.Reader to load the XML template without applying it to a specific document.

B.8 JCPrintEvent

Method	Description
getEventId()	Returns the eventId. The returned value is one of JCPrintEvent.BEGIN_PAGE, JCPrintEvent.END_PAGE, JCPrintEvent.OPEN_DOCUMENT, or JCPrintEvent.CLOSE_DOCUMENT,
getPageNumber()	Returns the page number or -1 if not applicable.

B.9 JCPrintListener

Method	Description
beginPage()	Invoked before a page has been printed.
closeDocument()	Invoked after a document has been printed.
endPage()	Invoked after a page has printed.
openDocument()	Invoked before a document has been printed.

B.10 JCTab

Method	Description
addTab()	Adds a tab to the style in the location specified.

Method	Description
setTabs()	Adds a list of identically aligned tabs to the style.

B.11 JCTextStyle

Method	Description
setLeftIndent()	Defines the amount of space between the left side of the frame and the left edge of every line in the paragraph except for the first line.
setParagraphIndent()	Defines the amount of space between the left side of the frame and the left edge of the first line in the paragraph.
setRightIndent()	Defines the amount of space between the right side of the frame and the right edge of the paragraph.

B.12 JCUnit.Margins

Method	Description
setBottom()	Sets the bottom margin to the width of the given JCUnit.Measure.
setLeft()	Sets the left margin to the width of the given JCUnit.Measure.
setRight()	Sets the right margin to the width of the given JCUnit.Measure.
setTop()	Sets the top margin to the width of the given JCUnit.Measure.

B.13 MathMatrix

Method	Description
getValueAt()	Retrieves the value at a particular row, column pair of index values in the matrix.
matrixValue()	Gets the contents of this MathValue as an array of Numbers.
numberValue()	Throws an UnsupportedOperationException.

Method	Description
`setValueAt()`	Sets the value at a particular row, column pair of index values in the matrix.
`toString()`	Outputs the value of this vector as a `String`.
`vectorValue()`	Throws an `UnsupportedOperationException`.

B.14 MathScalar

Method	Description
`matrixValue()`	Throws an `UnsupportedOperationException`.
`numberValue()`	Gets the contents of this `MathValue` as a `Number`.
`toString()`	Returns a String representation of this value.
`vectorValue()`	Throws an `UnsupportedOperationException`.

B.15 MathValue

Method	Description
`evaluate()`	Satisfies the `Expression` interface by returning the stored value. No evaluation is required because no operation is implied.
`getDataFormat()`	Retrieves the `NumberFormat` associated with this data.
`matrixValue()`	Gets the contents of this `MathValue` as a matrix of `Number`s.
`numberValue()`	Gets the contents of this `MathValue` as a `Number`.
`setDataFormat()`	Sets a `NumberFormat` to use on the contents of this `MathValue`.
`vectorValue()`	Gets the contents of this `MathValue` as a vector of `Number`s.

B.16 MathVector

Method	Description
`getValueAt()`	Retrieves the value at a particular index in the vector.

Method	Description
matrixValue()	Throws an UnsupportedOperationException.
numberValue()	Throws an UnsupportedOperationException.
setValueAt()	Sets the value at a particular index in the vector.
toString()	Outputs the value of this vector as a String.
vectorValue()	Gets the contents of this MathValue as a vector of Numbers.

Index