**FONT METRICS**

*Font metrics* are the measurements of characters in a particular font, which allow you to evenly space and uniformly align lines of text.

**Example**

Here are two lines of poetry appropriately spaced vertically and centered horizontally.

The graphics context that renders characters onto the computer display keeps the font metrics in an object of the API class `java.awt.FontMetrics`. This is an abstract class, which you can’t instantiate. You obtain the object by calling one of these methods within the `Graphics` object.

```java
FontMetrics getFontMetrics( )
// Get the font metrics of the Graphics object's current font.

FontMetrics getFontMetrics( Font f )
// Get the font metrics of the font passed as an argument.
```

**Example**

This code fragment shows the `paint` method of an applet. It sets a font for the graphics context at line 4 and retrieves the font metrics object at line 5.

```java
1 public void paint( Graphics g )
2 { 
3    super.paint( g );
4    g.setFont( new Font( "Helvetica", Font.PLAIN, 12 ) );
5    FontMetrics fm = g.getFontMetrics( );
6    . . .
7 }
```
You use the **FontMetrics** object to retrieve measurements of characters in that font. The names and meanings of these measurements vary widely in the typography and typesetting world, so those given here are specific to Java.

<table>
<thead>
<tr>
<th>Java Font Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(all measurements in pixels)</td>
</tr>
</tbody>
</table>

![Diagram showing FontMetrics](image)

- All measurements are relative to the **baseline**, which is the invisible line upon which a line of text sits.
- The distance a character extends above the baseline is called its **ascent**. Java keeps a single ascent value for the entire font, which is the typical height of capital letters (some characters may be taller).
- A character may descend below the baseline and that distance is called its **descent**. Java keeps a single descent value for the entire font, which is the lowest point of most characters (some may descend further).
- Java also keeps maximum ascent and descent values beyond which no character in the font is allowed to extend.
- The **leading** (pronounced “ledding”) is the distance between the descent of a line and the ascent of the line below it.
- The text **height** is the recommended distance between baselines and is calculated as the sum of the ascent, descent and leading.
- The **advance width** of a character or string is the width at its widest horizontal cross-section.
Here are the `FontMetric` methods that retrieve these measurements:

<table>
<thead>
<tr>
<th>Font Metric Methods of java.awt.FontMetrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getAscent( )</td>
</tr>
<tr>
<td>int getMaxAscent( )</td>
</tr>
<tr>
<td>int getDescent( )</td>
</tr>
<tr>
<td>int getMaxDescent( )</td>
</tr>
<tr>
<td>int getLeading( )</td>
</tr>
<tr>
<td>int getHeight( )</td>
</tr>
<tr>
<td>int charWidth( char ch )</td>
</tr>
<tr>
<td>int stringWidth( String s )</td>
</tr>
</tbody>
</table>

**Example**

Applet `FontMetricDemo` displays font metrics in its status bar.

```java
import java.applet.Applet;
import java.awt.*;

public class FontMetricDemo extends Applet {
    Font font = new Font( "Helvetica", Font.PLAIN, 16 );

    public void paint( Graphics g ) {
        super.paint( g );

        FontMetrics fm = g.getFontMetrics( font );
        showStatus( "Ascent:" + fm.getAscent( ) + "; " + "Descent:" + fm.getDescent( ) + "; " + "Leading:" + fm.getLeading( ) + "; " + "Height:" + fm.getHeight( ) + "; " + "CharWdt:" + fm.charWidth( 't' ) + "; " + "StrWdt:" + fm.stringWidth( "type" ) );
    }
}
```
Character Placement
Text is drawn onto a graphics context using the `drawString` method.

```java
void drawString( String s, int x, int y )
```

The latter two arguments give the \((x, y)\) position of the leftmost pixel of the string’s baseline. When displayed, the string extends above this position.

*Example*
```java
g.drawLine( 100, 72, 170, 72 );
g.drawString( "tYpe", 100, 72 );
```
**Horizontal Text Spacing**
The width metrics can be used to precisely position text horizontally.

**Example**
Pretend we want to display a mathematical formula using different fonts for the variables versus the numbers and operators, as shown to the right.

To do so, we must draw the items separately and switch the font of the graphics context before drawing each. The following picture illustrates the process:

```
2X + Y = 50
```

Since the drawing of each string involves the same three steps, shown at the left, this is an excellent use of a procedural abstraction. Thus, we code the steps as a method. The picture below shows how the method works when drawing the first piece of the formula.
Get the font metrics for the number/operator font
Set graphics context to the number/operator font
Draw the string "2" at x-position and y-position.
Add width of the string "2" onto x-position.

\[
\text{xPos} = \text{drawItem}( \text{g, nopFnt, "2", xPos, 26 });
\]

\[
\text{int drawItem( Graphics g, Font f, String s, int x, int y )}
\{
    \text{FontMetrics fm = g.getFontMetrics( f );}
    \text{g.setFont( f );}
    \text{g.drawString( s, x, y );}
    \text{return x + fm.stringWidth( s );}
\}
\]

The drawing is completed by calling the \textbf{drawItem} method for each of the five pieces of the formula. The complete applet follows.

```
import java.applet.Applet;
import java.awt. *

public class HorizontalSpacing extends Applet
{
    public void paint( Graphics g )
    {
        super.paint( g );
        int xPos = 10;
        xPos = drawItem( g, nopFnt, "2", xPos, 26 );
        xPos = drawItem( g, varFnt, "X", xPos, 26 );
        xPos = drawItem( g, nopFnt, " + ", xPos, 26 );
        xPos = drawItem( g, varFnt, "Y", xPos, 26 );
        xPos = drawItem( g, nopFnt, " = 50", xPos, 26 );
    }
}
```
int drawItem( Graphics g, Font f, String s, int x, int y )
// Draw string 's' in font 'f' onto graphics context 'g'.
// x, y give the position at which to draw.
// After drawing return the x-position for the next draw.
{  
    FontMetrics fm = g.getFontMetrics( f );  
    g.setFont( f );  
    g.drawString( s, x, y );  
    return x + fm.stringWidth( s );
}

Vertical Text Spacing
Spacing text vertically follows the same algorithmic process as that for horizontal spacing:

Set starting y-position
> Draw the string
> Add height of the string onto the y-position

The computations are much simpler, however, because the text height font metric doesn’t have to be recalculated for every line that is displayed. Furthermore, this metric is the recommended distance between baselines.

Example
Suppose we want to display the text shown at the right on equidistant lines. The distance is always equal to the text height metric.

```
One potato
Two potato
Three potato
Four
```
If we place the lines of text into an array then we can refine our algorithm accordingly.

<table>
<thead>
<tr>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Calculate font's text height
Set starting y-position the text height
Draw the string in array position \( k \)
Add height of the string onto the y-position
Increment \( k \)

The complete applet follows.

```java
import java.applet.Applet;
import java.awt.*;

public class VerticalSpacing extends Applet
{
    Font font = new Font( "Garamond", Font.ITALIC, 18 );
    String [] line = { "One potato", "Two potato",
                       "Three potato", "Four"       };

    public void paint( Graphics g )
    {
        super.paint( g );
        g.setFont( font );
        FontMetrics fm = g.getFontMetrics( );
        int textHeight = fm.getHeight( );
        int yPos = textHeight;
        for ( int k = 0; k < line.length; k++ )
        {
            g.drawString( line[k], 20, yPos );
            yPos += textHeight;
        }
    }
}
```
Centered Text
To center text you need to capture the GUI component’s width and height using these methods inherited by subclasses of `java.awt.Component`.

<table>
<thead>
<tr>
<th>Two Dimension Methods of java.awt.Component</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getWidth( )</code></td>
</tr>
<tr>
<td>// Return the width of this component in pixels.</td>
</tr>
<tr>
<td><code>int getHeight( )</code></td>
</tr>
<tr>
<td>// Return the height of this component in pixels.</td>
</tr>
</tbody>
</table>

These and the text height and width methods available from the `FontMetrics` class give you the data needed to center any text.

To determine the specific calculations needed, suppose \( w \) and \( h \) are the width and height of the component, respectively. \( tw \) and \( th \) are the width and height, respectively, of the text to be drawn.

The center point of the component is at \( \left( \frac{w}{2}, \frac{h}{2} \right) \). To be centered horizontally, the text’s baseline must start at \( x = \frac{w}{2} - \frac{tw}{2} \) so that half of it lies to the left of the component’s center point. To center it vertically, its baseline must start at \( y = \frac{h}{2} + \frac{th}{2} \).
Example
Let’s design the code necessary to center a single string $s$ within a GUI component. This code goes into the component’s `paint` method.

The following statements fetch the component’s dimensions:

```java
ing h = this.getHeight();
ing w = this.getWidth();
```

The width and height of string $s$ are calculated from a `FontMetrics` object:

```java
FontMetrics fm = g.getFontMetrics();
ing tw = fm.getStringWidth( s );
ing th = fm.getHeight();
```

The string is drawn using the calculations derived above:

```java
  g.drawString( s, w/2-tw/2, h/2+th/2 );
```

If the text is multi-lined, the total height can be calculated by multiplying $th$ by the number of lines. The total height can be used with the component height to determine the starting y-position for the first line of text. The remaining lines must be positioned as discussed in the section `Vertical Text Spacing` above. As each line of text is displayed, its width must be calculated separately so that its x-position can be calculated. Several of the exercises allow you to pursue these computations in Java.
Exercises

1. Enter the `FontMetricDemo` applet into jGRASP. Save, compile and run the applet. What are the six font metrics values that it displays in the status bar?

2. Modify `FontMetricDemo` so that it displays the metrics for this font:

   ```java
   Font( "Courier New", Font.BOLD, 20 )
   ```

   What are its six font metrics values? How can you tell that Courier New is a *monospaced* font? If you don’t know what a monospaced font is then Google it.

3. Modify `FontMetricDemo` so that it displays the metrics for this font:

   ```java
   Font( "Trebuchet MS", Font.PLAIN, 12 )
   ```

   What are its six font metrics values?

4. Modify the `HorizontalSpacing` applet by increasing the size of the font to 32 and making any other adjustments necessary to display the text properly.

5. Change the `HorizontalSpacing` applet so that it reads the font size from the user before creating the font. Make any other adjustments necessary to properly display the text no matter what size the user enters.

6. Change the `VerticalSpacing` applet so that it reads the font size from the user before creating the font. Make any other adjustments necessary to properly display the text no matter what size the user enters.

7. Write a Java applet to display this verse of Thomas Gray’s poem *Elegy Written in a Country Churchyard*. Follow these specifications:
   - Use the Trebuchet MS typeface, italics style and 16 point size.
   - Space the lines uniformly down the applet using the text height.
   - Leave one text height of space at the top.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>Using the code from the text centering example of page 10, write a Java applet that reads a one-line message from the user and displays it in the applet centered vertically and horizontally.</td>
</tr>
<tr>
<td>9.</td>
<td>Modify your solution to Exercise 7 so that each line is horizontally centered.</td>
</tr>
<tr>
<td>10.</td>
<td>Modify your solution to the previous exercise so that, in addition to each line being horizontally centered, the entire verse is vertically centered.</td>
</tr>
<tr>
<td>11.</td>
<td>Modify your solution to Exercise 6 so that each line is horizontally centered and all four lines are vertically centered.</td>
</tr>
<tr>
<td>12.</td>
<td>Modify your solution to Exercise 5 so that the line of text is centered vertically and horizontally.</td>
</tr>
</tbody>
</table>