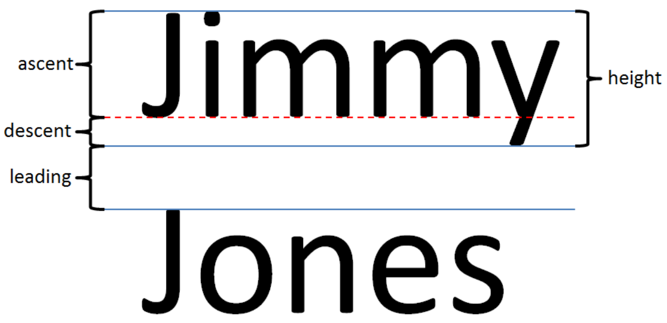
Text

Drawing text has some special properties and thus is treated in a separate chapter. We first need to talk about the sizing of text. Then we discuss fonts and how text is actually drawn. There is then the question of word wrapping and justification when drawing text. There are also some styling issues that describe how text should be drawn. Lastly we will touch on the international issues surrounding text.

Text metrics

Text sizing is described in points. A point is 1/72 of an inch. This claims tradition from typesetting, but the modern point size is a creation of digital printing. In early displays the resolution was 72 dots per inch making one point equal to one pixel. Screen resolutions have surpassed 72 dpi so the point/pixel relationship has also vanished. The reality is that except for very specialized uses the relative pixel/point size is ignored. We set the points to a size that visually appeals and forget it.

There are some important metrics that we must consider when working with text. Figure 4-1 shows the basics.



4-1 – Text metrics

Placement of text begins at the *baseline*, which is the dotted red line in figure 4-1. When styling text the size and font of the text may change, but the baseline is constant. That is where the reader's eye is expected to follow. The *ascent* is the distance from the baseline to the top of the tallest character in the font. The ascent is the same even if there are no tall characters in the string being displayed. The *descent* is the distance from the baseline to the lowest character in the font. Again whether any descending characters are being displayed is irrelevant. The *height* should be the ascent plus the descent. The *leading* is the distance between the descenders of one line that the ascenders of the line below. It is called leading because originally that space was created by inserting strips of lead between rows of font. In most systems the ascent and descent are part of the font's definition. For a given font and a size in points, we can ask for the ascent and descent for that font.

Leading is generally up to the software in deciding where to place the next line. Again we make our usual disclaimer that these general definitions do not apply to all systems. Sometimes the height includes leading for single spaced text. You need to check your system by drawing a few lines of texts with some guidelines to see how your system actually does this. Unfortunately many systems get a little vague in their documentation of these metrics. Try it first to be sure how it really works.

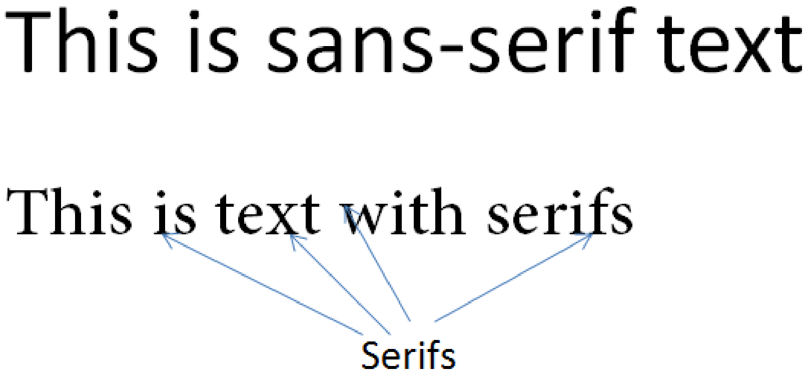
Fonts

Fonts are used for a variety of reasons. Fonts can change the style of the characters that are drawn. Fonts also relate strongly to the language or culture of the people reading the font. For our purposes a font is a mapping between a character code and a shape specification. Systems vary in whether they use quadratic curves (efficiency) or cubic curves (generality) in defining the shape of a character. A string is just an array of character codes. We take each character, use the font to look up a shape for that character and then draw the shape.

There are mono-spaced fonts and variable-spaced fonts. In a mono-spaced font, every character is the same width. This is the way typewriters used to work. We do not use mono-spaced fonts very often because they are harder to read and take up more horizontal screen space. In modern interactive systems, variable-spaced fonts are more common. In figure 4-1, notice that each character has a different width. The “m” is wider than everything else because it needs more pixels to be clearly read. The “i” is the narrowest character.

One of our needs is to calculate the width of a string of text or the horizontal position of a character within a string. In principle one should be able to inquire on the width of each character and then sum them up. In many systems that is exactly the right answer. Some systems differ in whether they include the space between characters or if that should be added later. Other systems take into account exactly where the characters are to be drawn before calculating that width. Virtually all Graphics objects have some mechanism for asking how wide a given string would be if it were drawn using the current settings. That is generally enough information for drawing and interaction.

An important characteristic of fonts is whether they are serif or sans-serif (without serifs). Figure 4-2 shows examples of both serif and sans-serif fonts. The serifs are the little feet on characters that are along the base line or along the lower-case reading line. When many characters are in a line, the serifs visually merge to create strong horizontal lines to guide the eye when reading. On the other hand, serifs add more detail to the characters and can make them individually more complicated to read. Serif fonts are used most often when there are large amounts of text to read. Sans-serif fonts are used in headings, posters and user interface labels where reading long lines of text are not frequent and the cleaner character figures make reading easier.



4-2 – Serif and sans-serif

Font storage
A font is basically a large collection of shapes. Historically the size of a font and the fact that they are shared among applications has meant that fonts are generally installed separately from the applications that use them. This has caused serious problems. If an application uses the "PigText" font and that application is run on a computer that does not have "PigText" installed then bad things happen. This is usually resolved in four ways.

The first is to ignore the problem and only use fonts that are likely to be on all machines, such as Time-Roman or Arial. This is easy, but slightly risky. It is also very limiting in the type-faces that you might use.

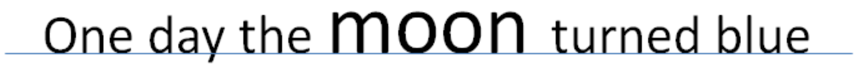
The second approach is to provide a "cascade" of fonts. For example one might specify "Tangerine, Times-Roman, serif" as my list of fonts. Tangerine is a nice script-like font that is frequently not available. By specifying a font list, the system can then try Times-Roman which does not have the style one might want but is a nice font, and quite common. If Times-Roman is not available then any serifed font can be picked. The point of the cascade of fonts to try is that we get exactly what we want if possible, but in the end we always get something that will work.

The third approach is internet-hosted fonts. If the font I want is posted somewhere on the Internet, then I can reference that font and have it downloaded on demand to any machine my application might run on. This means that font availability is not based on what is installed on the target computer. The risks here are that the application may need to run where internet access is not always available or that the site where the font is stored is temporarily or permanently down.

The last, and most reliable, approach is embed the font with the application itself. Because disk space is so cheap, the additional costs of having separate copies of a font in each application that uses the font are minimal. The embedding of a copy of a font is much less costly than failure of a program because of a missing font.

Drawing text

When drawing text there is generally a method on the Graphics object that expects a string and an (X,Y) point. In almost all drawing systems, the Y location defines where the baseline is to be drawn. The X location is generally the left-most edge of the first character of the string. By using Y as the baseline it makes it easy to draw several text segments that all align correctly. For example in figure 4-3 the sentence must be drawn using three different method calls, each using a different font size. The font size of the first segment ("One day the ") and the last segment ("turned blue") are the same but they are separated by a larger segment ("moon").



4-3 – Baseline alignment

The code to draw this sentence might be as follows:

```
void paint(Graphics g)
{   g.setFontSize(10);
    x = 10;
    g.drawString( x, 20, "One day the ");
    x += g.stringWidth("One day the ");
    g.setFontSize(20);
    g.drawString( x, 20, "moon ");
    x += g.stringWidth("moon ");
   g.setFontSize(10);
    g.drawString(x, 20, "turned blue");
}
```

As we draw each of the three segments we use the same Y value of 20 to keep their baselines all aligned and we advance X using the stringWidth() method to figure out how wide each segment should be.

Using stringWidth() we could figure out an X value for centering a string, but many systems allow one to change the meaning of X using an alignment property. Using Center alignment means that X is not the left edge but at the center of where the string is drawn. Similarly using Left alignment, X defines the right edge of the last character. Graphics systems vary a lot in the kinds of alignment that they support.

There are some graphics systems that support styled text (embedded instructions about size, bold, italic etc.) Such systems frequently support justification also. In those cases Y may be, not the baseline but the top of the text region. This is because the text system assumes that it will be handling any alignment issues.

Wrapping and justification

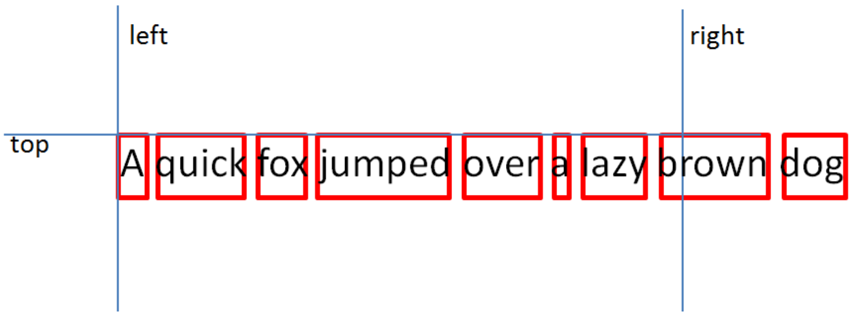
Graphics systems are mixed in terms of whether they perform word-wrapping and justification. Many systems simply provide a drawString() and stringWidth() method and expect programmers to work out their needs with those basic capabilities. Other systems provide full styling, word-wrapping and justification. One philosophy is to provide the basics from which anything is possible and the other is to support normal cases while providing as much support to the programmer as possible.

Regardless of what the graphics system provides, we need the ability to draw word-wrapped text and various forms of justification.

Let us assume that we have selected a font and we have a long string that will take many lines and we have a rectangle into which we want to draw that string. We will ignore the height of the rectangle and just draw until we run out of text.

Word wrap

Let us first consider the word wrapping problem and deal with justification later. Let us also assume that we have the three methods *drawString(left, base, string)*, *stringWidth(string)* and *fontHeight()*. We want to create a function *wrapText(left,top,right,text)* that will draw a string of text starting at top and staying between left and right. The string will be left justified. The problem is illustrated in figure 4-4.



4-4 – Word wrapping

A string of text is broken into words or other connected groups. Typically this is done using break characters such as white space, hyphens and punctuation. We then use stringWidth() to find the width of each group and use fontHeight() to find the height. Once we know the widths and the number of pixels we want to use for word spacing we can add them up left to right until we go beyond the right margin. We then back off one group and draw all of those groups. We use fontHeight() to locate the next line and continue from there. Many algorithms do not actually break up the text, but merely find the substrings and calculate the widths.

To draw the text we draw each group using drawString() with the same baseline. We start at *left* and add the width of the group we have drawn and our spacing.

Justification and centering

We can modify the above algorithm to right rather than left justify the text. Once we know that the word “A” through “lazy” will fit, we can calculate the total line width from the “A” to the “y”. If we want to right justify then we can start drawing the “A” at *(right-lineWidth)* rather than at left. That will have the “y” ending at *right*.

To center the text we again compute the width by summing up the spaces and the word groups. This time we start drawing at

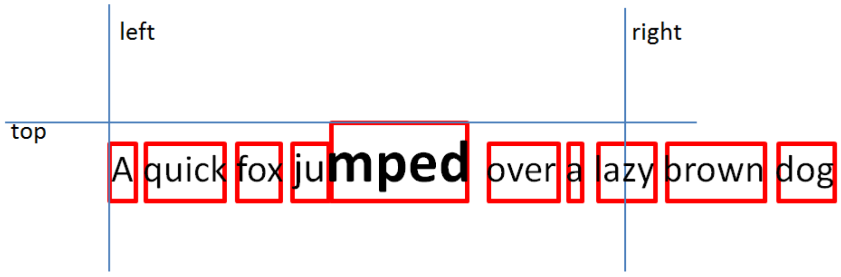
left+(right-left-lineWidth)/2

This will center the text.

To justify both margins we want to increase the spacing between groups. The amount of space to add is *right-left-lineWidth*. We can divide that by the number of breaks between groups (in our example there are 6 breaks.) This tells us the amount of space to add to each break so that “y” will end up at *right.* If we are using floating point coordinates, this will work. If we are using integer pixel coordinates the rounding or truncation will cut us short and we will need to adjust the algorithm slightly.

Text Styling

In figure 4-5 we see an example of text which has styling information embedded. Part of the word “jumped” uses a larger font and has been set to bold.



4-5 – drawing styled text

Drawing styled text requires more information than a simple string. The font-face, font-size, italic, bold, character spacing and other attributes may change in the course of the string. Note that the styling change for “mped” has increased the ascent and width of that piece of text.

Older systems developed various data structures for capturing styling information that would help programmers build up a description of the styling throughout the string. These were quite cumbersome and never very popular. A common technique now is to uses a subset of HTML to style the text. The text in figure 4-5 might be expressed as:

```
A quick fox ju<b><font size=”2”>mped</font></b> over a
lazy brown dog
```

The word wrapping algorithm for styled text is similar to that for plain text. In addition to its *width*, each group now has a *height* and an *ascent*. Note also that the word “jumped” has been split into two groups to accommodate the styling differences. However, there is no break between them. When drawing the text we need to find the maximum *ascent* and *descent* for all the groups that are to be drawn. The maximum *ascent* tells us where to place the baseline so that all of the groups align correctly regardless of their styling. The maximum *descent* along with the *leading* tells us where the top of the next line should go. Once the widths of the groups are known, justification and centering works the same as before.

Summary

The drawing of text is controlled by a font. A font is a set of shapes indexed by character code. The key metrics for strings are the ascent, descent, height, leading and width. For mono-spaced fonts the widths of the characters are the same. However, proportional fonts have different widths for each character which means we need the drawing system's help to find out the drawing width of a string of characters. Using this information we can perform word wrapping, right and left justification and centering for plain or styled text.