A letter is any character that the Java Character class considers to be either a “letter” or a “digit,” along with @ (at sign) and + (plus sign). Joiners include the Unicode M class, which matches a variety of nonspace and combining marks, other pieces of punctuation commonly found in URLs (since Wordle users expect to see URLs preserved as “words”), the apostrophe, and several characters used as apostrophes in the wild (such as U+2032, the PRIME character). Wordle accepts the tilde (~) as a word joiner but replaces it with a space in the output, thereby giving users an easy way to say “keep these words together” without having to find the magical key combination for a real nonbreaking space character.

**Determining the script**

Having extracted a list of words (whatever we take “word” to mean), we need to know how to display those words to the viewer. We first need to know what characters we’ll be expected to display, so that we can choose a font that supports those characters.

Wordle’s collection of fonts is organized in terms of what *scripts* each can support, where a script is what you might think of as an alphabet: a collection of *glyphs* that can be used to visually represent sequences of *characters* in one or more languages. A given script, in Unicode, is organized into one or more *blocks*. So, the task now is to determine which fonts the user might want to use by sniffing out which blocks are represented in the given text.

Java provides the static method `UnicodeBlock.of(int codePoint)` to determine which block a given code point belongs to. Wordle takes the most frequent words in a text and looks at the first character in each of those words. In the rather common case that the character is in the Latin block, we further check the rest of the word to see if it contains any Latin-1 Supplement characters (which would remove certain fonts from consideration) or any of the Latin Extended blocks (which would bar even more fonts). The most frequently seen block is the winner.

To keep it responsive and limit its use of network resources, Wordle is designed to permit the use of only one font at a time. A more full-featured word cloud might choose different fonts for different words; this could provide another visual dimension to represent, for example, different source texts.

As of this writing, Wordle supports the Latin, Cyrillic, Devanagari, Hebrew, Arabic, and Greek scripts. By design, Wordle does not support the so-called CJKV scripts, the scripts containing Chinese, Japanese, Korean, and Vietnamese ideographs. CJKV fonts are quite large and would take too long for the average Wordle user to download (and would cost a great deal in bandwidth charges). Also, determining word breaks for ideographic languages requires sophisticated machine-learning algorithms and large runtime data structures, which Wordle cannot afford.
**Unicode in a Nutshell**

Since Wordle understands text in Unicode terms, here's what you have to know in order to understand some of the terms and notations you'll see here.

The Unicode standard provides a universal coded character set and a few specifications for representing its characters in computers (as sequences of bytes).

A **character** is an abstract concept, meant to serve as an atom of written language. It is not the same thing as a “letter”—for example, some Unicode characters (accents, umlauts, zero-width joiners) are only meaningful in combination with other characters. Each character has a name (such as GREEK CAPITAL LETTER ALPHA) and a number of properties, such as whether it is a digit, whether it is an uppercase letter, whether it is rendered right-to-left, whether it is a diacritic, and so on.

A **character set** or **character repertoire** is another abstraction: it is an unordered collection of characters. A given character is either in, or not in, a given character set. Unicode attempts to provide a universal character set—one that contains every character from every written language in current and historical use—and the standard is constantly revised to bring it closer to that ideal.

A **coded character set** uniquely assigns an integer—a **code point**—to each character. Once you've assigned code points to the characters, you may then refer to those characters by their numbers. The convention used is an uppercase U, a plus sign, and a hexadecimal number. For example, the PRIME character mentioned earlier in this chapter has the code point U+2032.

Coded characters are organized according to the **scripts** in which they appear, and scripts are further organized into **blocks** of strongly related characters. For example, the Latin script (in which most European languages are written) is given in such blocks as Basic Latin (containing sufficient characters to represent Latin and English), Latin-1 Supplement (containing certain diacritics and combining controls), Latin Extended A, Latin Extended B, and so on.

When it comes time to actually put pixels onto a screen, a computer program interprets a sequence of characters and uses a **font** to generate **glyphs** in the order and location demanded by the context.

* See http://unicode.org.

---

**Guessing the language and removing stop words**

It would be neither interesting nor surprising to see that your text consists mostly of the words “the,” “it,” and “to.” To avoid a universe of boring Wordles, all alike, such **stop words** need to be removed for each recognized language. To know which list of stop words to remove for a given text, though, we have to guess what language that text is in.
Knowing the script is not the same as knowing the language, since many languages might use the same script (e.g., French and Italian, which share the Latin script).

Wordle takes a straightforward approach to guessing a text’s language: it selects the 50 most frequent words from the text and counts how many of them appear in each language’s list of stop words. Whichever stop word list has the highest hit count is considered to be the text’s language.

How do you create a list of stop words? As with the definition of a “word,” described earlier, this kind of thing is a matter of taste, not science. You typically start by counting all of the words in a large corpus and selecting the most frequently used words. However, you might find that certain high-frequency words add a desirable flavor to the output while other, lower-frequency words just seem to add noise, so you may want to tweak the list a bit.

Many of Wordle’s stop word lists came from users who wanted better support for their own languages. Those kind folks are credited on the Wordle website.

By default Wordle strips the chosen language’s stop words from the word list before proceeding to the next steps, but Wordle users can override this setting via a menu checkbox.

**Assigning weights to words**

Wordle takes the straight path in assigning a numeric *weight* to each word. The formula is: weight = word count.

**Layout**

Once you’ve analyzed your text, you’re left with a list of words, each of which has some numeric *weight* based on its frequency in the text. Wordle normalizes the weights to an arbitrary scale, which determines the magnitude of various constants that affect the resulting image (such as the minimum size of a hierarchical bounding box leaf, as described later in this chapter). You’re now ready to turn words into graphical objects and to position those objects in space.

**Weighted words into shapes**

For each word, Wordle constructs a font with a point size equal to the word’s scaled weight, then uses the font to generate a Java2D Shape (see Example 3-2).
Example 3-2. How to turn a String into a Shape

```java
private static final FontRenderContext FRC = new FontRenderContext(null, true, true);

public Shape generate(final Font font, final double weight, final String word, final double orientation) {
    final Font sizedFont = font.deriveFont((float) weight);
    final char[] chars = word.toCharArray();
    final int direction = Bidi.requiresBidi(chars, 0, chars.length) ? Font.LAYOUT_RIGHT_TO_LEFT : Font.LAYOUT_LEFT_TO_RIGHT;
    final GlyphVector gv = sizedFont.layoutGlyphVector(FRC, chars, 0, chars.length, direction);
    Shape result = gv.getOutline();
    if (orientation != 0.0){
        result = AffineTransform.getRotateInstance(orientation).createTransformedShape(result);
    }
    return result;
}
```

The playing field

Wordle estimates the total area to be covered by the finished word cloud by examining the bounding box for each word, summing the areas, and adjusting the sum to account for the close packing of smaller words in and near larger words. The resulting area is proportioned to match the target aspect ratio (which is, in turn, given by the dimensions of the Wordle applet at the moment of layout).

The constants used to adjust the size of the playing field, the area in which Wordles are laid out, were determined by the time-honored tradition of futzing around with different numbers until things looked “good” and worked “well.” As it happens, the precise size of the playing field is rather important, because the field boundaries are used as constraints during layout. If your playing field is too small, your placement will run slowly and most words will fall outside the field, leaving you with a circle (because once a word can’t be placed on the field, Wordle relaxes that constraint and you wind up with everything randomly distributed around some initial position). If it’s too large, you’ll get an incoherent blob (because every nonintersecting position is acceptable).

One “gotcha” to look out for is an especially long word, which could have a dimension far larger than the calculated width or height based on area. You must make sure that your playing field is big enough to contain the largest word, at least.

Remember that the playing field is an abstract space, a coordinate system not corresponding to pixels, inches, or any other unit of measurement. In this abstract space, you can lay out the word shapes and check for intersections. When it comes time to actually put pixels on the screen, you can do some scaling into screen units.
Placement

Having created a place to put words, it’s time to position the words in that space. The overall placement strategy is a randomized greedy algorithm in which words are placed, one at a time, on the playing field. Once a word is placed, its position does not change.

Wordle offers the user a choice of placement strategies. These strategies influence the shape and texture of the completed Wordle, by determining where each word “wants” to go. On the Wordle website, the choices are center-line and alphabetical center-line. Both strategies place words near the horizontal center-line of the playing field (not necessarily upon it, but scattered away from it by a random distribution). The alphabetical strategy sorts the words alphabetically and then distributes their preferred x coordinates across the playing field.

Interesting effects are possible through the use of smarter placement strategies. For example, given clustering data—information about which words tend to be used near each other—the placement strategy can make sure that each word tries to appear near the last word from its cluster that was placed on the field (see Figure 3-16).

Figure 3-16. The result of a clustering placement strategy
The word shapes are sorted by their respective weights, in descending order. Layout proceeds as in Example 3-3, with the result as illustrated in Figure 3-17.

**Example 3-3. The secret Wordle algorithm revealed at last!**

For each word $w$ in sorted words:
placementStrategy.place($w$)

while $w$ intersects any previously placed words:
  move $w$ a little bit along a spiral path

![Figure 3-17. The path taken by the word "Denmark"](image)

To make matters a bit more complicated, Wordle optionally tries to get the words to fit entirely within the rectangular boundaries of the playing field—this is why it’s important to guess how big the whole thing is going to be. If the rectangular constraint is turned on, the intersection-handling routine looks like Example 3-4.

**Example 3-4. Constraining words to the playing field**

while $w$ intersects any previously placed words:
  do {
    move $w$ a little bit along a spiral path
  } while any part of $w$ is outside the playing field and the spiral radius is still smallish