Data 2: Text

This unit introduces code elements for working with language.

Syntax introduced:
`char, String`

I SENSE THE SUN IN THE STREET,
ALL SPACE IN THE STREET.
BANG! THE SUN HAS SLID.

This poem was generated from software written by Margaret Masterman and was featured in the 1968 Cybernetic Serendipity exhibition at the Institute of Contemporary Arts (ICA) in London. This exhibition exposed the public to examples of software-generated poems, music, and drawings. While the poem may or may not conform to your ideas about great poetry, the exhibition was important for its early emphasis on using the computer as a language processing machine. A common misconception holds that computer programming is applicable only to technical fields. While there is a strong connection between programming and technology, it's not the only realm in which computers can make for interesting collaborators. Programming can be approached with an emphasis on language, making computers potentially interesting to a far broader audience.

Some of the earliest explorations of the computer outside scientific research focused on software as a language engine. The history of artificial intelligence (AI) has a strong component of language processing. John McCarthy's LISP programming language made processing text easy and became popular for early experimentation in AI. The controversial ELIZA software, written by Joseph Weizenbaum in 1966, parodies the dialog between a Rogerian therapist and a patient by rephrasing the patient's statements as questions. People input statements through a keyboard and the software constructs a reply. For example, if the patient types "I feel depressed," ELIZA might respond, "Why do you say you are depressed?" Terry Winograd's SHRDLU project, c. 1970, used the same kind of interaction between keyboard input and text response, but it earnestly explored the computer's potential for understanding natural language. SHRDLU made it possible for a person to have a discussion with the computer about an arrangement of simulated blocks. For example, to the query "How many blocks are not in the box?" the software would respond "Four of them" based on the current status of the blocks.

Researchers have continued to explore language as an interface with and input to software. Emerging software services such as automated translation and speech-to-text conversions are not always reliable, but they are fascinating to explore. For example, if we take two simple English sentences . . .

Translation requires nuance. Can it be performed by a machine?

... and convert them to Italian using an online translation service, we are given this text:

La traduzione richiede la sfumatura. Può essere effettuata da una macchina?

If we take the Italian translation and convert it back to English, we now have

The translation demands the shading. Can be carried out from one machine?

Similarly, software designed to convert spoken language into written language has its limitations. Both technologies, however, can be used in controlled circumstances as unique ways of working with text and software.

This unit does not discuss artificial intelligence or language parsing, but text is one of the most common types of data created and modified by software. The text created for Email, publications, and Web pages is a vast resource of data that can be stored and presented through the data types and functions introduced below.

Characters

The `char` data type stores typographic symbols such as *A*, *d*, *5*, and *$*. The name *char* is short for *character*, and this type of data is distinguished from other typographic symbols in the program by surrounding single quotes. `Char` variables are declared and assigned in the same way as the `int` and `float` types.

```
char a = 'n';          // Assign 'n' to variable a                    11-01
char b = n;            // ERROR! Without quotes, n is a variable
char c = "n";          // ERROR! The "" defines n as a String, not a char
char d = 'not';        // ERROR! The char type can hold only one character
```

The following example creates a new `char` variable, assigns values, and prints the values to the console.

```
char letter = 'A'; // Declare variable letter and assign 'A'            11-02
println(letter);   // Prints "A" to the console
letter = 'B';      // Assign 'B' to variable letter
println(letter);   // Prints "B" to the console
```

Many characters have a corresponding number on the standardized ASCII table. For example, *A* is 65, *B* is 66, *C* is 67, etc. You can find which character matches which number by looking at an ASCII table (such as in Appendix C, p. 664) or by testing with the `println()` function:

```
char letter = 'A'; // Declare variable letter and assign 'A'            11-03
println(letter);   // Prints "A" to the console
int n = letter;    // Assign the numerical value of 'A' to variable n
println(n);        // Prints "65" to the console
```

Appendix C also includes information about using non-ASCII characters (for instance, a character with an accent or an umlaut) or characters from non-Roman alphabets such as Japanese or Korean.

The mapping between numeric and alphabetic formats emphasizes the importance of data types. The following program prints the letters *A* to *Z* to the console by incrementing the `char` variable in a `for` structure.

```
char letter = 'A';  // Declare variable letter and assign 'A'                11-04
for (int i = 0; i < 26; i++) {
  print(letter);    // Prints a character to the console
  letter++;         // Add 1 to the value of the character
}
println('.');       // Adds a period to the end of the alphabet
```

Words, Sentences

Use the `String` data type to store words and sentences. Surrounding double quotes distinguish strings from characters and the rest of the program. Quotation marks define "s" as a string, while single quotes (apostrophes) define 's' as a character, and without either it could be a variable name. The `String` data type is different from the data types `int`, `float`, and `char` because it is an *object* (p. 395), a composite data type containing multiple data elements and functions. The previously introduced data types `PImage` and `PFont` are also objects. `String` variables are declared and assigned in the familiar way, but the word `String` must be capitalized:

```
String a = "Eponymous";  // Assign "Eponymous" to a                          11-05
String b = 'E';          // ERROR! The '' define E as a char
String c = "E";          // Assign "E" to c
string d = "E";          // ERROR! String must be capitalized
```

The following example demonstrates some basic ways to use this data type:

```
// The String data type can contain long and short text elements            11-06
String s1 = "Rakete bee bee?";
String s2 = "Rrrrrrrrrrrrrrrrummmmmpffff tillffff tooooo?";
println(s1);  // Prints "Rakete bee bee?"
println(s2);  // Prints "Rrrrrrrrrrrrrrrrummmmmpffff tillffff tooooo?"

// Strings can be combined with the + operator
String s3 = "Rakete ";
String s4 = "rinnzekete";
String s5 = s3 + s4;
println(s5);  // Prints "Rakete rinnzekete"
```

If you have a large quantity of text to display in your programs, it's better to load the text into the program from a file than to store it in `String` variables. This process is explained in Input 6 (p. 427). The `char` and `String` data types will be used to more interesting ends in the proceeding units on Typography and Input. There are many functions inside the `String` data type for operating on text. They perform actions such as making all the letters lowercase or looking only at one letter within the text. These functions are explained in the next unit.

Exercises

1. *Create five `char` variables and assign a character to each. Write each to the console.*
2. *Create two `String` variables and assign a word to each. Write each to the console.*
3. *Store a sentence in a `String` and write it to the console.*

Data 3: Conversion, Objects

This unit introduces converting values from one data format to another and working with data as objects.

Syntax introduced:

```
boolean(), byte(), char(), int(), float(), str()
"." (dot operator)
PImage.width, PImage.height
String.length(), String.startsWidth(), String.endsWidth(),
String.charAt(), String.toCharArray(), String.subString(),
String.toLowerCase(), String.toUpperCase()
String.equals()
```

When a variable is created, its data type is specified. If the variable will store numeric data, the `int` or `float` types are used. If the variable will store character data, a `String` can be used to store multiple characters, or the `char` data type can store just one. A `true` or `false` value is stored in a `boolean` variable, an image is stored in a `PImage` variable, and a typeface is stored in a `PFont` font variable. After the variable is created, it can only be assigned data elements of its type. Sometimes, though, it's necessary to convert a value from one type of data to another, a task for which Processing has several functions.

The data types `int`, `float`, `boolean`, and `char` are called *primitive* data types because they store a single data element. The types `String`, `PImage`, and `PFont` are different. Variables created from these data types are *objects*. Objects are usually composed of several primitive data types (or other objects), and can also have functions inside to act on their data. For example, a `String` object stores an array of characters and has functions that return the number of characters or the character at a specific location. Objects are visually distinguished from primitive data types with capitalization.

Data conversion

Some data type conversions are automatic and others need to be made explicit with functions written for data type conversion. Automatic conversions are made between compatible types. For example, an `int` can be automatically converted to a `float`, but a `float` can't be automatically converted to an `int`:

12-01

```
float f = 12.6;
int i = 127;
f = i;  // Converts 127 to 127.0
i = f;  // Error: Can't automatically convert a float to an int
```

How does one know which data types are compatible and which require an explicit conversion? Conversions that involve a loss of information must be explicit. When converting an `int` to a `float`, nothing is lost. When converting a `float` to an `int`, however, the numbers after the decimal point are lost. Explicit conversions are a way of stating in code that this loss of information is intentional. The functions for explicit data type conversion are `boolean()`, `byte()`, `char()`, `float()`, `int()`, and `str()`. Each is used to convert other data types to the type for which the function is named.

The `boolean()` function converts the number 0 to `false` and all other numbers to `true`. It converts the string "true" to `true` and the string "false" to `false`.

```
int i = 0;                                                    12-0
boolean b = boolean(i);  // Assign false to b
int n = 12;
b = boolean(n);          // Assign true to b
String s = "false";
b = boolean(s);          // Assign false to b
```

The `byte()` function converts other types of data to a byte representation. A `byte` can only be a whole number between -128 and 127; therefore, when a number outside this range is converted, its value wraps to the corresponding byte representation.

```
float f = 65.0;                                               12-0
byte b = byte(f);  // Assign 65 to b
char c = 'E';
b = byte(c);       // Assign 69 to b
f = 130.0;
b = byte(f);       // Assign -126 to b
```

The `char()` function converts other types of data to a character representation. An explanation of the numbering can be found in Appendix C (p. 664).

```
int i = 65;                                                   12-0
byte y = 72.0;
char c = char(i);  // Assign 'A' to c
c = char(y);       // Assign 'H' to c
```

The `float()` function converts other types of data to a floating-point representation. It is most often used when making calculations. As discussed in Math 1 (p. 43), dividing two integers will always evaluate as an integer, which is a problem when working with fractions. For example, when the integer number 3 is divided by 6, the answer is the integer value 0 rather than the often desired floating-point value 0.5. Converting one of these values to a `float` allows the expression to evaluate to a floating-point value.

```
int i = 2;                                                    12-05
int j = 3;
float f1 = i/j;              // Assign 0.0 to f1
float f2 = i/float(j);  // Assign 0.6666667 to f2
```

The `int()` function converts other types of data to an integer representation. Many of the math functions only return `float` values, and it's necessary to convert them to integers for use in other parts of a program.

```
float f = 65.3;                                               12-06
int i = int(f);  // Assign 65 to i
char c = 'E';
i = int(c);          // Assign 69 to i
```

The `str()` function converts other types of data to a string representation:

```
int i = 3;                                                    12-07
String s = str(i);  // Assign "3" to s
float f = -12.6;
s = str(f);              // Assign "-12.6" to s
boolean b = true;
s = str(b);              // Assign "true" to s
```

The `nf()` function (p. 422) provides more control when converting an `int` or a `float` to a `String`. It can set the number of decimal places and pad the number with zeros.

Objects

Variables created with the `PImage`, `PFont`, and `String` data types are *objects*. Variables within an object are called *fields*, and functions within an object are called *methods*. Fields and methods are accessed with the *dot operator*, a period placed between the name of the object and the name of a data element or function inside the object. The `PImage`, `PFont`, and `String` data types each have their own unique additional data elements and functions.

The `PImage` data type has two fields that store the width and height of the image, named, appropriately, `width` and `height`. To access these, write the name of the object followed by a *dot* and the name of the variable:

```
PImage img = loadImage("ohio.jpg");  // Load a 320 x 240 pixel image   12-08
int w = img.width;   // Assign 320 to w
int h = img.height;  // Assign 240 to h
println(w);  // Prints "320"
println(h);  // Prints "240"
```

The `width` and `height` variables can only be read—assigning a value to them will cause problems. The image's `width` and `height` values can be used to position images side by side or to place shapes in relation to an image. The `PImage` object has many functions for manipulating the pixels of an image. They are discussed in Image 3 (p. 321), Image 4 (p. 347), and Image 5 (p. 355).

The `String` data type includes methods for examining individual characters within the string, extracting parts of strings, converting an entire string to uppercase or lowercase characters, and comparing two `String` variables. Some of the most common `String` methods are introduced below, and more are discussed in the Processing reference.

The `length()` method returns the number of characters in a `String` object:

```
String s1 = "Player Piano";                                    12-09
String s2 = "P";
println(s1.length());  // Prints "12"
println(s2.length());  // Prints "1"
```

Notice the difference in syntax between the array field `length` (p. 304) and the `String` method `length()`. They both calculate the number of elements in their object, but because the technique for getting the number of elements in a `String` is a method, the parentheses are necessary.

The `startsWith()` and `endsWith()` methods test whether a string starts or ends with the string used as the parameter:

```
String s1 = "Slaughterhouse Five";                             12-10
println(s1.startsWith("S"));     // Prints "true"
println(s1.startsWith("Five")); // Prints "false"
println(s1.endsWith("Five"));    // Prints "true"
```

The `charAt()` method is used to read a single character within a string. This method has one parameter to define the character that is returned.

```
String s = "Verde";                                             12-11
println(s.charAt(0)); // Prints "V"
println(s.charAt(2)); // Prints "r"
println(s.charAt(4)); // Prints "e"
```

The `toCharArray()` method creates an array of characters from the contents of a string.

```
String s = "Azzurro";                                           12-12
char[] c = s.toCharArray();
println(c[0]); // Prints "A"
println(c[1]); // Prints "z"
```

The `String` method `substring()` returns a new string that is a part of the original. When the method is used with one parameter, the string is read from the position given as the parameter to the end of the string. When two parameters are used, the string between the two parameter positions is returned.

12-13
```
String s = "Giallo";
println(s.substring(2));      // Prints "allo"
println(s.substring(4));      // Prints "lo"
println(s.substring(1, 4));   // Prints "ial"
println(s.substring(0, s.length()-1));  // Prints "Giall"
```

The `String` method `toLowerCase()` returns a copy of the string with all of the characters made lowercase. The method `toUpperCase()` does the same for uppercase.

12-14
```
String s = "Nero";
println(s.toLowerCase());  // Prints "nero"
println(s.toUpperCase());  // Prints "NERO"
```

Because the `String` data type is an object, it's not possible to compare two strings with relational operators. Using == to compare two objects will compare only whether they are stored in the same location in memory, not their actual contents. Instead, the `equals()` method is used to determine whether two `String` variables contain the same characters.

12-15
```
String s1 = "Bianco";
String s2 = "Bianco";
String s3 = "Nero";
println(s1.equals(s2));  // Prints "true"
println(s1.equals(s3));  // Prints "false"
```

This unit introduced ways to utilize the additional data and functions within objects. These methods will be useful in the following units, but the full potential of working with objects is not revealed until Structure 4 (p. 395), which presents the techniques for writing your own objects.

Exercises

1. *Write a program to convert the value of an integer to other data types. Display the conversions in the console.*
2. *Load an image and display its height and width to the console using the `PImage` fields.*
3. *Explore the `String` methods and use one or more of them to reconfigure two sentences into one variable.*

Typography 1: Display

This unit introduces loading and setting fonts and displaying letters on screen.

Syntax introduced:

```
PFont, loadFont(), textFont(), text()
textSize(), textLeading(), textAlign(), textWidth()
```

The evolution of typographic reproduction and display technologies has and continues to impact human culture. Early printing techniques developed by Johannes Gutenberg in fifteenth-century Germany using positionable letters cast from lead provided a catalyst for increased literacy and the scientific revolution. Automated typesetting machines, such as the Linotype invented in the nineteenth century, changed the way information was produced, distributed, and consumed. In the digital era, the way we consume text has changed drastically since the proliferation of personal computers in the 1980s and the rapid growth of the Internet in the 1990s. Text from Emails, websites, and instant messages fill computer screens, and while many of the typographic rules of the past apply, type on screen requires additional considerations for the sake of communication and legibility.

Letters on screen are created by setting the color of pixels. The quality of the typography is constrained by the resolution of the screen. Because screens have a low resolution in comparison to paper, techniques have been developed to enhance the appearance of type on screen. The fonts on the earliest Apple Macintosh computers were comprised of small bitmap images created at specific sizes like 10, 12, and 24 points. Using this technology, a variation of each font was designed for each size of a particular typeface. For example, the character *A* in the San Francisco typeface used a different image to display the character at size 12 and 18. When the LaserWriter printer was introduced in 1985, Postscript technology defined fonts with a mathematical description of each character's outline. This allowed type on screen to scale to large sizes and still look smooth. Apple and Microsoft later developed TrueType, another outline font format. More recently, these technologies were merged into the OpenType format. In the meantime, methods to smooth black-and-white text on screen were introduced. These anti-aliasing techniques use gray pixels at the edge of characters to compensate for low screen resolution.

The proliferation of personal computers in the mid-1980s spawned a period of rapid typographic experimentation. Digital typefaces are software, and the old rules of metal and photo type no longer apply. The Dutch typographers known as LettError explain, "The industrial methods of producing typography meant that all letters had to be identical . . . Typography is now produced with sophisticated equipment that doesn't impose such rules. The only limitations are in our expectations."[1] LettError expanded the possibilities of typography with their typeface Beowolf (p. 169). It prints every letter differently so that each time an *A* is printed, for example, it will have a different

shape. During this time, typographers such as Zuzana Licko and Barry Deck began creating radical and innovative typefaces with the assistance of new software tools. The flexibility of software has enabled extensive font revivals and historic homages such as Adobe Garamond from Robert Slimbach and The Proteus Project from Jonathan Hoefler. Typographic nuances such as ligatures—connections between letter pairs such as fi, ff, and æ—made impractical by modern mechanized typography are flourishing again through software font tools.

Loading fonts, Drawing text

Before letters can be displayed on the screen with Processing, a font must first be converted into the VLW format. To convert a font, select the "Create Font" option from the Tools menu. A window opens and displays the names of the fonts installed on your computer that can be converted. Select a font from the list and click "OK." The font generates and is copied into the current sketch's *data* folder. To make sure the font is there, click on the Sketch menu and select "Show Sketch Folder."

Like the early Macintosh fonts, the VLW format used by Processing stores each letter of the alphabet as an image. The VLW format is a quick way to render text and makes it possible to include a font with a sketch. A font created at size 12 will therefore have a smaller file size than a font stored at size 96 because the images require less space. The Create Font dialog box offers the option to set the size of the font and to select whether it will be smooth (antialiased). This box also offers the option to export "All Characters," which means every character in the font will be included. The name of the file can also be changed before the font is created.

After the font is created, drawing letters to the display window is a multistep process. Before a font is used in a program, it must be loaded and set as the current font. Processing has a unique data type called `PFont` to store font data. Make a new variable of the type `PFont` and use the `loadFont()` function to load the font. The `textFont()` function must be used to set the current font. The `text()` function is used to draw characters to the screen:

```
text(data, x, y)
text(stringdata, x, y, width, height)
```

The *`data`* parameter can be a `String`, `char`, `int`, or `float`. The *`stringdata`* parameter can only be a `String`. The *`x`* and *`y`* parameters set the position of the lower-left corner. The optional *`width`* and *`height`* parameters set boundaries. The `text()` function draws the characters at the current font's original size. The `fill()` function controls the color and transparency of text. This function affects text the same way it affects shapes such as `rect()` and `ellipse()`. Text is not affected by `stroke()`.

The following examples use a font named Ziggurat. To run these examples, you will need to use the "Create Font" tool to create your own font. Change the name of the parameter of `loadFont()` to the name of the font that you created.

13-01

```
PFont font;  // Declare the variable
font = loadFont("Ziggurat-32.vlw");  //  Load the font
textFont(font);  // Set the current text font
fill(0);
text("LAX", 0, 40);  // Write "LAX" at coordinate (0,40)
text("AMS", 0, 70);  // Write "AMS" at coordinate (0,70)
text("FRA", 0, 100);  // Write "FRA" at coordinate (0,100)
```

13-02

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
text(19, 0, 36);    // Write 19 at coordinate (0,36)
text(72, 0, 70);    // Write 72 at coordinate (0,70)
text('R', 62, 70);  // Write 'R' at coordinate (62,70)
```

13-03

```
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
fill(0);
String s = "Response is the medium";
text(s, 10, 20, 80, 50);
```

13-04

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(255);  // White
text("DAY", 0, 40);
fill(0);    // Black
text("CVG", 0, 70);
fill(102);  // Gray
text("ATL", 0, 100);
```

13-05

```
PFont font;
font = loadFont("Ziggurat-72.vlw");
textFont(font);
fill(0, 160);  // Black with low opacity
text("1",  0, 80);
text("2", 15, 80);
text("3", 30, 80);
text("4", 45, 80);
text("5", 60, 80);
```

To use two fonts in one program, create two `PFont` variables and use the `textFont()` function to change the current font.

13-06

```
PFont font1, font2;
font1 = loadFont("Ziggurat-32.vlw");
font2 = loadFont("ZigguratItalic-32.vlw");
fill(0);
// Set the font to Ziggurat-32.vlw
textFont(font1);
text("GNU", 6, 45);
// Set the font to ZigguratItalic-32.vlw
textFont(font2);
text("GNU", 2, 80);
```

Text attributes

Processing includes functions to control the way text is displayed—for example, by changing its size, leading (the spacing between lines), and alignment. Processing can also calculate the width of any character or group of characters, a useful function for arranging shapes and typographic elements.

Fonts in Processing are images and not vector outlines. When the font is drawn at a different size from the size at which it was created, it is scaled and therefore does not always look as crisp and smooth. For example, if a font is created at 12 pixels and is displayed at 96 pixels, it will appear blurry. The `textSize()` function sets the current font size:

```
textSize(size)
```

The *`size`* parameter defines the dimension of the letters in units of pixels.

13-07

```
// Reducing a font created at 32 pixels
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
text("LNZ", 0, 40);   // Large
textSize(18);
text("STN", 0, 75);   // Medium
textSize(12);
text("BOS", 0, 100);  // Small
```

13-08

```
// Enlarging a font created at 12 pixels
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
textSize(32);
fill(0);
text("LNZ", 0, 40);   // Large
textSize(18);
text("STN", 0, 75);   // Medium
textSize(12);
text("BOS", 0, 100);  // Small
```

The `textLeading()` function sets the spacing between lines of text:

textLeading(dist)

The *dist* parameter defines this space in units of pixels.

13-09

```
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
String lines = "L1 L2 L3";
textLeading(10);
fill(0);
text(lines, 5, 15, 30, 100);
textLeading(20);
text(lines, 36, 15, 30, 100);
textLeading(30);
text(lines, 68, 15, 30, 100);
```

Letters and words can be drawn from their center, left, and right edges. The `textAlign()` function sets the alignment for drawing text:

textAlign(MODE)

The *MODE* parameter can be `LEFT`, `CENTER`, or `RIGHT`. It sets the display characteristics of the letters in relation to the value of the *x* parameter used in the `text()` function.

The settings for `textSize()`, `textLeading()`, and `textAlign()` will be used for all subsequent calls to the `text()` function. However, note that the `textSize()` function will reset the text leading, and the `textFont()` function will reset both the size and the leading.

13-10

```
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
line(50, 0, 50, 100);
fill(0);
textAlign(LEFT);
text("Left", 50, 20);
textAlign(RIGHT);
text("Right", 50, 40);
textAlign(CENTER);
text("Center", 50, 80);
```

The `textWidth()` function calculates and returns the pixel width of any character or text string. This number is calculated from the current font and size as defined by the `textFont()` and `textSize()` functions. Because the letters of every font are a different size and letters within many fonts have different widths, this function is the only way to know how wide a string or character is when displayed on screen. For this reason, always use `textWidth()` to position elements relative to text, rather than hard-coding them into your program.

13-11

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
char c = 'U';
float cw = textWidth(c);
text(c, 22, 40);
rect(22, 42, cw, 5);
String s = "UC";
float sw = textWidth(s);
text(s, 22, 76);
rect(22, 78, sw, 5);
```

Exercises

1. *Explore different typefaces in Processing. Draw your favorite word to the display window in your favorite typeface.*
2. *Draw a paragraph of text to the display window. Carefully select the composition.*
3. *Use two different typefaces to display the dialog between two characters.*

Notes

1. Ellen Lupton, *Thinking with Type: A Critical Guide for Designers, Writers, Editors, & Students* (Princeton Architectural Press, 2004), p. 29.

Input 2: Keyboard

This unit introduces keyboard input.

Syntax introduced:

```
keyPressed, key, keyCode
```

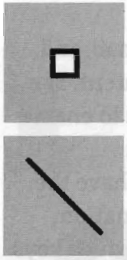
Keyboards are typically used to input characters for composing documents, Email, and instant messages, but the keyboard has potential for use beyond its original intent. The migration of the keyboard from typewriter to computer expanded its function to enable launching software, moving through the menus of software applications, and navigating 3D environments in games. When writing your own software, you have the freedom to use the keyboard data any way you wish. For example, basic information such as the speed and rhythm of the fingers can be determined by the rate at which keys are pressed. This information could control the speed of an event or the quality of motion. It's also possible to ignore the characters printed on the keyboard itself and use the location of each key relative to the keyboard grid as a numeric position.

The modern computer keyboard is a direct descendant of the typewriter. The position of the keys on an English-language keyboard is inherited from early typewriters. This layout is called QWERTY because of the order of the top row of letter keys. It was developed for typewriters to put physical distance between frequently typed letter pairs, helping reduce the likelihood of the typebars colliding and jamming as they hit the ribbon. There are variations on this layout for different languages including the AWERTY layout for the French language and the QWERTZ layout for German. The alphabetic differences are small, but the symbol placement on these keyboard variations can be extreme. For example, commonly used programming symbols such as { and } are not printed on French keyboards, but can be accessed through the Alt Gr key. Keyboards for languages with different alphabets often keep the same physical key arrangement but replace the characters with, for example, Greek, Arabic, or Thai characters. Keyboards for alphabets with thousands of characters, such as Chinese, don't have a direct mapping between key and character—they use a system where a series of key presses is interpreted by the operating system to build each symbol.

In recent years we've seen the dominance of keyboards challenged by alternate input methods on small, handheld devices such as the Palm Pilot. Although the Palm's Graffiti software makes it simple to translate hand gestures into characters, the mini-keyboards on mobile phones and other personal digital assistants have reconfirmed many people's preference for inputting data with a keyboard. Some users have become adept at typing characters with a mobile-phone keypad, while others opt for phones with miniature QWERTY keyboards. On the other hand, speech recognition is always improving and provides an alternative to the keyboard for certain kinds of tasks.

Keyboard data

Processing registers the most recently pressed key and whether a key is currently pressed. The `boolean` variable `keyPressed` is `true` if a key is pressed and is `false` if not. Including this variable in an `if` structure allows lines of code to run only if a key is pressed. The `keyPressed` variable remains `true` while the key is held down and becomes `false` only when the key is released.



25-01

```
// Draw a rectangle while any key is pressed

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  if (keyPressed == true) {  // If the key is pressed,
    line(20, 20, 80, 80);    // draw a line
  } else {                   // Otherwise,
    rect(40, 40, 20, 20);    // draw a rectangle
  }
}
```

25-02

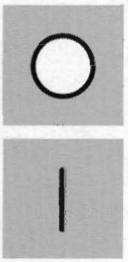
```
// Move a line while any key is pressed

int x = 20;

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  if (keyPressed == true) {  // If the key is pressed
    x++;                     // add 1 to x
  }
  line(x, 20, x-60, 80);
}
```

The key variable is of the char data type and stores the most recently pressed key. The key variable can store only one value at a time. The most recent key pressed will be the only value stored in the variable. A key can be displayed on screen by loading a font and using the text() function (p. 112).

25-03

```
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("ThesisMonoLight-72.vlw");
  textFont(font);
}

void draw() {
  background(0);
  text(key, 28, 75);
}
```

The key variable may be used to determine whether a specific key is pressed. The following example uses the expression key == 'A' to test if the *A* key is pressed. The single quotes signify *A* as the data type char. The expression key == "A" will cause an error because the double quotes signify the *A* as a String, and it's not possible to compare a String with a char. The logical AND symbol, the && operator, is used to connect the expression with the keyPressed variable to ascertain that the key pressed is the uppercase *A*.



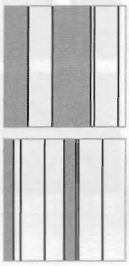
25-04

```
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  // If the 'A' key is pressed draw a line
  if ((keyPressed == true) && (key == 'A')) {
    line(50, 25, 50, 75);
  } else {  // Otherwise, draw an ellipse
    ellipse(50, 50, 50, 50);
  }
}
```

If you want to check for both uppercase and lowercase letters, you have to extend the relational expression with a logical OR, the || relational operator. Line 10 in the previous program would be changed to

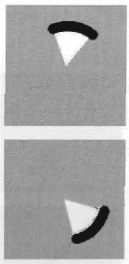
```
if ((keyPressed == true) && ((key == 'a') || (key == 'A'))) {
```

Because each character has a numeric value as defined by the ASCII table (p. 665), the value of the key variable can be used to control visual attributes such as the position and color of shape elements.



25-05

```
int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  if (keyPressed == true) {
    x = key - 32;
    rect(x, -1, 20, 101);
  }
}
```

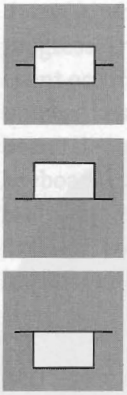


25-06

```
float angle = 0;

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(8);
}

void draw() {
  background(204);
  if (keyPressed == true) {
    if ((key >= 32) && (key <= 126)) {
      // If the key is alphanumeric,
      // convert its value into an angle
      angle = map(key, 32, 126, 0, TWO_PI);
    }
  }
  arc(50, 50, 66, 66, angle-PI/6, angle+PI/6);
}
```

Coded keys

In addition to reading key values for numbers, letters, and symbols, Processing can also read the values from other keys including the arrow keys and the Alt, Control, Shift, Backspace, Tab, Enter, Return, Escape, and Delete keys. The variable `keyCode` stores the `ALT`, `CONTROL`, `SHIFT`, `UP`, `DOWN`, `LEFT`, and `RIGHT` keys as constants. Before determining which coded key is pressed, it's necessary to check first to see if the key is coded. The expression `key == CODED` is `true` if the key is coded and `false` otherwise. Even though not alphanumeric, the keys included in the ASCII (p. 664) specification (`BACKSPACE`, `TAB`, `ENTER`, `RETURN`, `ESC`, and `DELETE`) will not be identified as a coded key. If you're making cross-platform projects, note that the Enter key is commonly used on PCs and UNIX and the Return key is used on Macintosh. Check for both Enter and Return to make sure your program will work for all platforms (see code 26-08).



25-07

```
int y = 35;

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  line(10, 50, 90, 50);
  if (key == CODED) {
    if (keyCode == UP) {
      y = 20;
    } else if (keyCode == DOWN) {
      y = 50;
    }
  } else {
    y = 35;
  }
  rect(25, y, 50, 30);
}
```

Exercises

1. *Use the number keys on the keyboard to modify the movement of a line.*
2. *Create a typing program to display a different image for each letter on the keyboard.*
3. *Use the arrow keys to change the position of a shape within the display window.*

Input 5: Time, Date

This unit introduces using the current time and date as inputs.

Syntax introduced:
`second(), minute(), hour(), millis(), day(), month(), year()`

Humans have a relative perception of time, but machines attempt to keep precise, regular time. Previous civilizations used sundials and water clocks to visualize the passage of time; today, most people use the digital numeric clock and the twelve-hour circular clock with a minute, second, and hour hand. Each of these representations reflects their technology. A numeric digital readout is appropriate for a digital timekeeping mechanism in need of an inexpensive display. A timekeeping mechanism built from circular gears lends itself to a circular presentation of time. The tower-sized clocks of the past with enormous gears and weights have evolved and shrunk into devices so inexpensive and abundant that digital devices such as microwave ovens, mobile phones, and computers all display the time and date.

Reading the current time and date into a program opens an interesting area of exploration. Knowledge of the current time makes it possible to write a program that changes its colors every day depending on the date, or a digital clock that plays an animation every hour. The ability to input time and date information enables software tools for remembering, reminding, and informing and creates the potential for whimsical time-based events.

Seconds, Minutes, Hours

Processing programs can read the value of the computer's clock. The current second is read with the `second()` function, which returns values from 0 to 59. The current minute is read with the `minute()` function, which also returns values from 0 to 59. The current hour is read with the `hour()` function, which returns values in the 24-hour time notation from 0 to 23. In this system midnight is 0, noon is 12, 9:00 a.m. is 9, and 5:00 p.m. is 17. Run this program to see the current time:

28-01

```
int s = second();  // Returns values from 0 to 59
int m = minute();  // Returns values from 0 to 59
int h = hour();    // Returns values from 0 to 23
println(h + ":" + m + ":" + s);  // Prints the time to the console
```

Placing these functions inside `draw()` allows the time to be read continuously. This example reads the current time and updates the text area with the passage of each second:

28-02

```
int lastSecond = 0;

void setup() {
  size(100, 100);
}

void draw() {
  int s = second();
  int m = minute();
  int h = hour();
  // Only prints once when the second changes
  if (s != lastSecond) {
    println(h + ":" + m + ":" + s);
    lastSecond = s;
  }
}
```

You can create a clock with a numerical display by using the `text()` function to draw the numbers to the display window. The `nf()` function (p. 422) is used to space the numbers equally from left to right. Single digit numbers are padded on their left with a zero so all numbers occupy a two-digit space at all times.

28-03

```
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("Pro-20.vlw");
  textFont(font);
}

void draw() {
  background(0);
  int s = second();
  int m = minute();
  int h = hour();
  // The nf() function spaces the numbers nicely
  String t = nf(h,2) + ":" + nf(m,2) + ":" + nf(s,2);
  text(t, 10, 55);
}
```

There are many different ways to express the passage of time. In the next example, horizontal lines mark the current second, hour, and minute. The left edge of the display window is 0 and the right edge is the maximum for each time component. Because the values from the time functions range from 0 to 59 and from 0 to 23, they are modified to

all have the range of 0 to 99. Each time value is divided by its maximum value and then multiplied by 100.0.

28-04

```
void setup() {
  size(100, 100);
  stroke(255);
}

void draw() {
  background(0);
  float s = map(second(), 0, 60, 0, 100);
  float m = map(minute(), 0, 60, 0, 100);
  float h = map(hour(), 0, 24, 0, 100);
  line(s, 0, s, 33);
  line(m, 34, m, 66);
  line(h, 67, h, 100);
}
```

These normalized time values can also be used to simulate the second, minute, and hour hands on a traditional clock face. In this case, the values are multiplied by 2π to display the time as points around a circle. Because the `hour()` function returns values in 24-hour time, the program converts the hour value to 12-hour time scale by calculating the hour % 12 (the % symbol is introduced on p. 45).

28-05

```
void setup() {
  size(100, 100);
  stroke(255);
}

void draw() {
  background(0);
  fill(80);
  noStroke();
  // Angles for sin() and cos() start at 3 o'clock;
  // subtract HALF_PI to make them start at the top
  ellipse(50, 50, 80, 80);
  float s = map(second(), 0, 60, 0, TWO_PI) - HALF_PI;
  float m = map(minute(), 0, 60, 0, TWO_PI) - HALF_PI;
  float h = map(hour() % 12, 0, 12, 0, TWO_PI) - HALF_PI;
  stroke(255);
  line(50, 50, cos(s) * 38 + 50, sin(s) * 38 + 50);
  line(50, 50, cos(m) * 30 + 50, sin(m) * 30 + 50);
  line(50, 50, cos(h) * 25 + 50, sin(h) * 25 + 50);
}
```

In addition to reading the current time, each Processing program counts the time passed since the program started. This time is stored in milliseconds (thousandths of a second). Two thousand milliseconds is 2 seconds, and 200 milliseconds is 0.2 seconds. This number is obtained with the `millis()` function and can be used to trigger events and calculate the passage of time:

28-06

```
// Uses millis() to start a line in motion three seconds
// after the program starts

int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  if (millis() > 3000) {
    x++;
  }
  line(x, 0, x, 100);
}
```

The `millis()` function returns an `int`, but it is sometimes useful to convert it to a `float` that represents the number of seconds elapsed since the program started. The resulting number can be used to control the sequence of events in an animation.

28-07

```
int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  float sec = millis() / 1000.0;
  if (sec > 3.0) {
    x++;
  }
  line(x, 0, x, 100);
}
```

Date

Date information is read in a similar way as the time. The current day is read with the `day()` function, which returns values from 1 to 31. The current month is read with the `month()` function, which returns values from 1 to 12 where 1 is January, 6 is June, and 12 is December. The current year is read with the `year()` function, which returns the four-digit integer value of the present year. Run this program to see the current date in the console:

28-08

```
int d = day();    // Returns values from 1 to 31
int m = month();  // Returns values from 1 to 12
int y = year();   // Returns four-digit year (2007, 2008, etc.)
println(d + " " + m + " " + y);
```

The following example checks to see if it is the first day of the month and prints the message "Welcome to a new month." to the console if it is the first day of the month.

28-09

```
void draw() {
  int d = day();  // Values from 1 to 31
  if (d == 1) {
    println("Welcome to a new month.");
  }
}
```

The following example runs continuously and checks to see if it is New Year's Day. If so, the message "Today is the first day of the year!" is printed to the console.

28-10

```
void draw() {
  int d = day();   // Values from 1 to 31
  int m = month(); // Values from 1 to 12
  if ((d == 1) && (m == 1)) {
    println("Today is the first day of the year!");
  }
}
```

Exercises

1. *Make a simple clock to run an animation for two seconds at the beginning of each minute.*
2. *Create an abstract clock that communicates the passage of time through graphical quantity rather than numerical symbols.*
3. *Write a program to draw images to the display window corresponding to specific dates (e.g., display a pumpkin on Halloween).*

Typography 2: Motion

This unit introduces typography in motion.

Despite the potential for kinetic type within film, animated typography didn't begin to flourish until the 1950s with the film title work of Saul Bass, Maurice Binder, and Pablo Ferro. These designers and their peers set a high standard with their kinetic title sequences for films such as *North by Northwest* (1959), *Dr. No* (1962), and *Bullitt* (1968). They explored the evocative power of kinetic letterforms to set a mood and express additional layers of meaning in relation to written language. In subsequent years the design of film titles has matured and been augmented by experimental typography for television and the Internet.

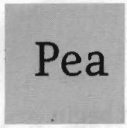
Software has played a large role in extending the possibilities of type in motion. The Visual Language Workshop (VLW), founded by Muriel Cooper at the MIT Media Lab in 1985, applied rigorous design thinking to the presentation of kinetic and spatial typography. Researchers including Suguru Ishizaki, Lisa Strausfeld, Yin Yin Wong, and David Small produced progressive typographic explorations ranging from the expression of animated phrases to the navigation of vast typographic landscapes. Because programs did not exist to perform these experiments, the researchers developed custom software to realize their ideas. While at the VLW, David Small created the *Talmud Project* to explore reading in a unique way. It displays the Talmud and related commentaries on the screen simultaneously. A dial controls the legibility of each text through blurring and fading, while keeping each source in context. Peter Cho, building on the explorations of the VLW, wrote software that continued to push the boundaries of expressive kinetic typography. His *Letterscapes* website presents every letter of the Roman alphabet as a character with a unique motion and response in relation to its form. With his *Takeluma* project, Cho went even further by inventing a kinetic alphabet for visualizing speech.

In the last decade, many software tools have been released that facilitate working with kinetic typography. Adobe's Flash software has provided new freedom for working with type on the Web, and Adobe After Effects has supported more sophisticated typography in film and television. This unit introduces techniques for exploring kinetic typography with code.

Words in motion

For typography to move, the program must run continuously, and therefore it requires a `draw()` function. Using typography within `draw()` requires three steps. First, a `PFont` variable must be declared outside of `setup()` and `draw()`. Next, the font should be loaded and set within `setup()`. Finally, the font can be used to place characters on the screen inside `draw()` with the `text()` function.

The following examples use a font named Eureka. To run these examples, you will need to use the "Create Font" tool to create your own font. Change the name of the parameter to `loadFont()` to the name of the font that you created.




```
PFont font;
String s = "Pea";

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  fill(0);
}

void draw() {
  background(204);
  text(s, 22, 20);
}
```

To put type into motion, simply draw it at a different position each frame. Words can move in an orderly fashion if their position is changed slightly each frame, and they can move without apparent order if placed in an arbitrary position each frame.


```
PFont font;
float x1 = 0;
float x2 = 100;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  fill(0);
}

void draw() {
  background(204);
  text("Right", x1, 50);
  text("Left", x2, 100);
  x1 += 1.0;
  if (x1 > 100) { x1 = -150; }
  x2 -= 0.8;
  if (x2 < -150) { x2 = 100; }
}
```

36-03

```
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  noStroke();
}

void draw() {
  fill(204, 24);
  rect(0, 0, width, height);
  fill(0);
  text("flicker", random(-100, 100), random(-20, 120));
}
```

Typography need not move in order to change over time. More subtle transformations, such as changes in the gray value or transparency of the text, can be made by changing the value of a variable within `draw()`.

36-04

```
PFont font;
int opacity = 0;
int direction = 1;

void setup() {
  size(100, 100);
  font = loadFont("EurekaSmallCaps-36.vlw");
  textFont(font);
}

void draw() {
  background(204);
  opacity += 2 * direction;
  if ((opacity < 0) || (opacity > 255)) {
    direction = -direction;
  }
  fill(0, opacity);
  text("fade", 4, 60);
}
```

Applying the transformations `translate()`, `scale()`, and `rotate()` can also create motion.

36-05

```
PFont font;
String s = "VERTIGO";
float angle = 0.0;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-90.vlw");
  textFont(font, 24);
  fill(0);
}

void draw() {
  background(204);
  angle += 0.02;
  pushMatrix();
  translate(33, 50);
  scale((cos(angle/4.0) + 1.2) * 2.0);
  rotate(angle);
  text(s, 0, 0);
  popMatrix();
}
```

Another technique, called rapid serial visual presentation (RSVP), displays words on the screen sequentially and provides a fundamentally different way to think about reading. Run this program and change the frame rate to see how it affects the process of reading. To store the words within one variable called *words*, this example uses a data element called an Array (explained in Data 4, p. 301).

36-06

```
PFont font;
String[] words = { "Three", "strikes", "and", "you're",
                   "out...", " " };
int whichWord = 0;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-32.vlw");
  textFont(font);
  textAlign(CENTER);
  frameRate(4);
}
```

```
  void draw() {
    background(204);
    whichWord++;
    if (whichWord == words.length) {
      whichWord = 0;
    }
    text(words[whichWord], width/2, 55);
  }
```

36-06 cont.

Letters in motion

Individually animated letters offer more flexibility than entire moving words. Building words letter by letter, each with a different movement or speed, can convey a particular meaning or tone. Working in this way requires more patience and often longer programs, but the results can be more rewarding because of the increased possibilities.

36-07

```
// The size of each letter grows and shrinks from
// left to right

PFont font;
String s = "AREA";
float angle = 0.0;

void setup() {
  size(100, 100);
  font = loadFont("EurekaMono-48.vlw");
  textFont(font);
  fill(0);
}

void draw() {
  background(204);
  angle += 0.1;
  for (int i = 0; i < s.length(); i++) {
    float c = sin(angle + i/PI);
    textSize((c + 1.0) * 32 + 10);
    text(s.charAt(i), i*26, 60);
  }
}
```


```
// Each letter enters from the bottom in sequence and
// stops when it reaches its destination

PFont font;
String word = "rise";
char[] letters;
float[] y;                    // Y-coordinate for each letter
int currentLetter = 0;        // Letter currently in motion

void setup() {
  size(100, 100);
  font = loadFont("EurekaSmallCaps-36.vlw");
  textFont(font);
  letters = word.toCharArray();
  y = new float[letters.length];
  for (int i = 0; i < letters.length; i++) {
    y[i] = 130;               // Position off the screen
  }
  fill(0);
}

void draw() {
  background(204);
  if (y[currentLetter] > 35) {
    y[currentLetter] -= 3; // Move current letter up
  } else {
    if (currentLetter < letters.length-1) {
      currentLetter++;        // Switch to next letter
    }
  }
  // Calculate x to center the word on screen
  float x = (width - textWidth(word)) / 2;
  for (int i = 0; i < letters.length; i++) {
    text(letters[i], x, y[i]);
    x += textWidth(letters[i]);
  }
}
```

Exercises

1. *Select a noun and an adjective. Animate the noun to reveal the adjective.*
2. *Use the transformation functions to animate a short phrase.*
3. *Select a verb and animate each letter of the word to convey its meaning.*

Typography 3: Response

This unit introduces typography that responds to input from the mouse and keyboard.

Many people spend hours a day inputting letters into computers, but this action is very constrained. What features could be added to a text editor to make it more responsive to the typist? For example, the speed of typing could decrease the size of the letters, or a long pause in typing could add many spaces, mimicking a person's pause while speaking. What if the keyboard could register how hard a person is typing (the way a piano plays a soft note when a key is pressed gently) and could automatically assign attributes such as *italics for soft presses* and **bold for forceful presses**? These analogies suggest how conservatively current software treats typography and typing.

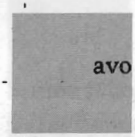
Many artists and designers are fascinated with type and have created unique ways of exploring letterforms with the mouse, keyboard, and more exotic input devices. A minimal yet engaging example is John Maeda's *Type, Tap, Write* software, created in 1998 as an homage to manual typewriters. This software uses the keyboard as the input to a black-and-white screen representation of a keyboard. Pressing the number keys cause the software to cycle through different modes, each revealing a playful interpretation of keyboard data. Casey Reas and Golan Levin's *Dakadaka* software from 2000, named after the sounds made while hitting a keyboard, explores the percussive and rhythmic aspects of typing. Input from the keyboard is translated into four different positional abstract alphabets that change according to the speed of typing and the order of the pressed keys. In Jeffrey Shaw and Dirk Groeneveld's *The Legible City* (1989–91), buildings are replaced with three-dimensional letters to create a city of typography that conforms to the streets of a real place. In the Manhattan version, for instance, texts from the mayor, a taxi driver, and Frank Lloyd Wright comprise the city. The image is presented on a projection screen, and the user navigates by pedaling and steering a stationary bicycle situated in front of the projected image. Projects such as these demonstrate that software presents an extraordinary opportunity to extend the way we read and write.

Responsive words

Typographic elements can be assigned behaviors that define a personality in relation to the mouse or keyboard. A word can express aggression by moving quickly toward the mouse, or one moving away slowly can express timidity.

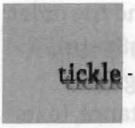
avoid

```
// The word "avoid" stays away from the mouse because its
// position is set to the inverse of the cursor position

PFont f;

void setup() {
  size(100, 100);
  f = loadFont("Eureka-24.vlw");
  textFont(f);
  textAlign(CENTER);
  fill(0);
}

void draw() {
  background(204);
  text("avoid", width-mouseX, height-mouseY);
}
```



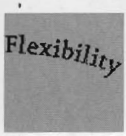
tickle

```
// The word "tickle" jitters when the cursor hovers over

PFont f;
float x = 33;  // X-coordinate of text
float y = 60;  // Y-coordinate of text

void setup() {
  size(100, 100);
  f = loadFont("Eureka-24.vlw");
  textFont(f);
  noStroke();
}

void draw() {
  fill(204, 120);
  rect(0, 0, width, height);
  fill(0);
  // If the cursor is over the text, change the position
  if ((mouseX >= x) && (mouseX <= x+55) &&
      (mouseY >= y-24) && (mouseY <= y)) {
    x += random(-5, 5);
    y += random(-5, 5);
  }
  text("tickle", x, y);
}
```



Responsive letters

Breaking a word into its component letters creates more options in determining its response to the mouse or keyboard. Independent letters that each have the ability to respond in a different way contribute to the word's total response. The following two examples demonstrate this technique. The `toCharArray()` method (p. 108) is used to extract the individual characters from a `String` variable and put them into an array of characters. The `charAt()` method (p. 108) is an alternate way to isolate the individual letters within a String.



37-03

```
// The horizontal position of the mouse determines the
// rotation angle. The angle accumulates with each letter
// drawn to make the typography curve.

String word = "Flexibility";
PFont f;
char[] letters;

void setup() {
  size(100, 100);
  f = loadFont("Eureka-24.vlw");
  textFont(f);
  letters = word.toCharArray();
  fill(0);
}

void draw() {
  background(204);
  pushMatrix();
  translate(0, 33);
  for (int i = 0; i < letters.length; i++) {
    float angle = map(mouseX, 0, width, 0, PI/8);
    rotate(angle);
    text(letters[i], 0, 0);
    // Offset by the width of the current letter
    translate(textWidth(letters[i]), 0);
  }
  popMatrix();
}
```

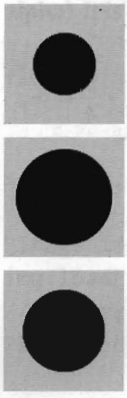

```
// Calculates the size of each letter based on the
// position of the cursor so the letters are larger
// when the cursor is closer

String word = "BULGE";
char[] letters;
float totalOffset = 0;
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  letters = word.toCharArray();
  textAlign(CENTER);
  fill(0);
}

void draw() {
  background(204);
  translate((width - totalOffset) / 2, 0);
  totalOffset = 0;
  float firstWidth = (width / letters.length) / 4.0;
  translate(firstWidth, 0);
  for (int i = 0; i < letters.length; i++) {
    float distance = abs(totalOffset - mouseX);
    distance = constrain(distance, 24, 60);
    textSize(84 - distance);
    text(letters[i], 0, height - 2);
    float letterWidth = textWidth(letters[i]);
    if (i != letters.length-1) {
      totalOffset = totalOffset + letterWidth;
      translate(letterWidth, 0);
    }
  }
}
```

Exercises

1. *Change the visual attributes of a word as the cursor moves across the display window.*
2. *Draw a verb on screen and have it respond to the cursor to communicate its meaning.*
3. *Select an adverb and a verb. Design the way the verb responds to the mouse to communicate the adverb.*

Image as data

This unit has introduced digital images as one-dimensional sequences of numbers that define colors. This numerical data, however, need not be viewed as colors—it can be used to generate motion or define the vertices of a shape. The following examples use the data from the `pixels[]` array of an image to generate alternative representations.



40-14

```
// Convert pixel values into a circle's diameter

PImage arch;
int index;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
  color c = arch.pixels[index];  // Get a pixel
  float r = red(c) / 3.0;  // Get the red value
  ellipse(width/2, height/2, r, r);
  index++;
  if (index == width*height) {
    index = 0;  // Return to the first pixel
  }
}
```

40-15

```
// Convert the red values of pixels to line lengths

PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
```

```
  int my = constrain(mouseY, 0, 99);
  for (int i = 0; i < arch.height; i++) {
    color c = arch.pixels[my*width + i];  // Get a pixel
    float r = red(c);  // Get the red value
    line(i, 0, i, height/2 + r/6);
  }
}
```

40-15 cont.

```
// Convert the blue values from one row of the image
// to the coordinates for a series of lines

PImage arch;

void setup() {
  size(100, 100);
  smooth();
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
  int mx = constrain(mouseX, 0, arch.width-1);
  int offset = mx * arch.width;
  beginShape(LINES);
  for (int i = 0; i < arch.width; i += 2) {
    float r1 = blue(arch.pixels[offset + i]);
    float r2 = blue(arch.pixels[offset + i + 1]);
    float vx = map(r1, 0, 255, 0, height);
    float vy = map(r2, 0, 255, 0, height);
    vertex(vx, vy);
  }
  endShape();
}
```

40-16

Exercises

1. *Write your own image filter by modifying the values of* `pixels[]`.
2. *Explore different kernels to convolve an image and write a program to display your most interesting discovery.*
3. *Load an image and use its data to generate an animation that reflects the original image.*