Structure 1: Code Elements

This unit introduces the most basic elements and vocabulary for writing software.

Syntax introduced:
```
// (comment), /* */ (multiline comment)
";" (statement terminator), "," (comma)
print(), println()
```

Creating software is an act of *writing*. Before starting to write code, it's important to acknowledge the difference between writing a computer program and writing an Email or an essay. Writing in a human language allows the author to utilize the ambiguity of words and to have great flexibility in constructing phrases. These techniques allow multiple interpretations of a single text and give each author a unique voice. Each computer program also reveals the style of its author, but there is far less room for ambiguity. While people can interpret vague meanings and can usually disregard poor grammar, computers cannot. Some of the linguistic details of writing code are discussed here to prevent early frustration. If you keep these details in mind as you begin to program, they will gradually become habitual. This unit presents variations of a simple program that sets the size and background color of the display window, demonstrating some of the most basic elements of writing code with Processing.

Comments

Comments are ignored by the computer but are important for people. They let you write notes to yourself and to others who read your programs. Because programs use symbols and arcane notation to describe complex procedures, it is often difficult to remember how individual parts of a program work. Good comments serve as reminders when you revisit a program and explain your thoughts to others reading the code. Commented sections appear in a different color than the rest of the code. This program explains how comments work:

1-01
```
// Two forward slashes are used to denote a comment.
// All text on the same line is a part of the comment.
// There must be no spaces between the slashes. For example,
// the code "/ /" is not a comment and will cause an error

// If you want to have a comment that is many
// lines long, you may prefer to use the syntax for a
// multiline comment
```

```
/*                                                                   1-01
  A forward slash followed by an asterisk allows the                 cont.
  comment to continue until the opposite
*/

// All letters and symbols that are not comments are translated
// by the compiler. Because the following lines are not comments,
// they are run and draw a display window of 200 x 200 pixels
size(200, 200);
background(102);
```

Functions

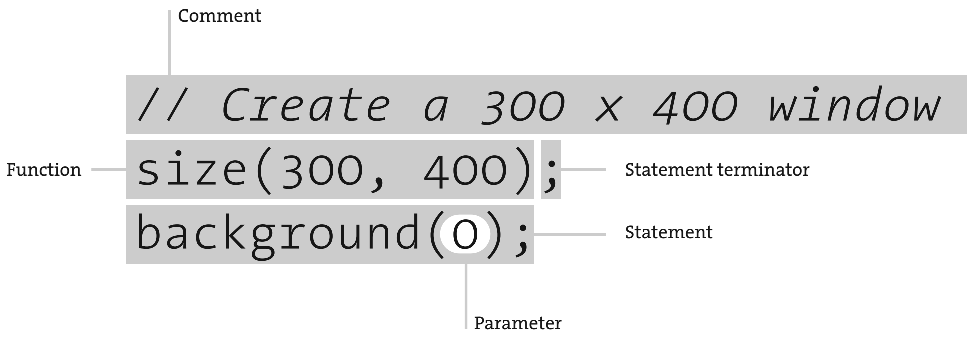
Functions allow you to draw shapes, set colors, calculate numbers, and to execute many other types of actions. A function's name is usually a lowercase word followed by parentheses. The comma-separated elements between the parentheses are called parameters, and they affect the way the function works. Some functions have no parameters and others have many. This program demonstrates the `size()` and `background()` functions.

```
// The size function has two parameters. The first sets the width    1-02
// of the display window and the second sets the height
size(200, 200);

// This version of the background function has one parameter.
// It sets the gray value for the background of the display window
// in the range of 0 (black) to 255 (white)
background(102);
```

Expressions, Statements

Using an analogy to human languages, a software expression is like a phrase. Software expressions are often combinations of operators such as +, *, and / that operate on the values to their left and right. A software expression can be as basic as a single number or can be a long combination of elements. An expression always has a value, determined by evaluating its contents.

Expression	*Value*
5	5
122.3+3.1	125.4
((3+2)*-10) + 1	-49



Anatomy of a program
Every program is composed of different language elements. These elements work together to describe the intentions of the programmer so they can be interpreted by a computer. The anatomy of a more complicated program is shown on page 176.

Expressions can also compare two values with operators such as > (greater than) and < (less than). These comparisons are evaluated as `true` or `false`.

Expression	*Value*
`6 > 3`	`true`
`54 < 50`	`false`

A set of expressions together create a statement, the programming equivalent of a sentence. It's a complete unit that ends with the statement terminator, the programming equivalent of a period. In the Processing language, the statement terminator is a semicolon.

Just as there are different types of sentences, there are different types of statements. A statement can define a variable, assign a variable, run a function, or construct an object. Each will be explained in more detail later, but examples are shown here:

1-03

```
size(200, 200);   // Runs the size() function
int x;            // Declares a new variable x
x = 102;          // Assigns the value 102 to the variable x
background(x);    // Runs the background() function
```

Omitting the semicolon at the end of a statement, a very common mistake, will result in an error message, and the program will not run.

Case sensitivity

In written English, some words are capitalized and others are not. Proper nouns like Ohio and John and the first letter of every sentence are capitalized, while most other words are lowercase. In many programming languages, some parts of the language must be capitalized and others must be lowercase. Processing differentiates between uppercase and lowercase characters; therefore, writing “Size” when you mean to write “size” creates an error. You must be exacting in adhering to the capitalization rules.

1-04

```
size(200, 200);
Background(102);  // ERROR! The B in "background" is capitalized
```

Whitespace

In many programming languages, including Processing, there can be an arbitrary amount of space between the elements of a program. Unlike the rigorous syntax of statement terminators, spacing does not matter. The following two lines of code are a standard way of writing a program:

1-05

```
size(200, 200);
background(102);
```

However, the whitespace between the code elements can be set to any amount and the program will run exactly the same way:

1-06

```
size

(   200,
  200)            ;
background   (          102)
         ;
```

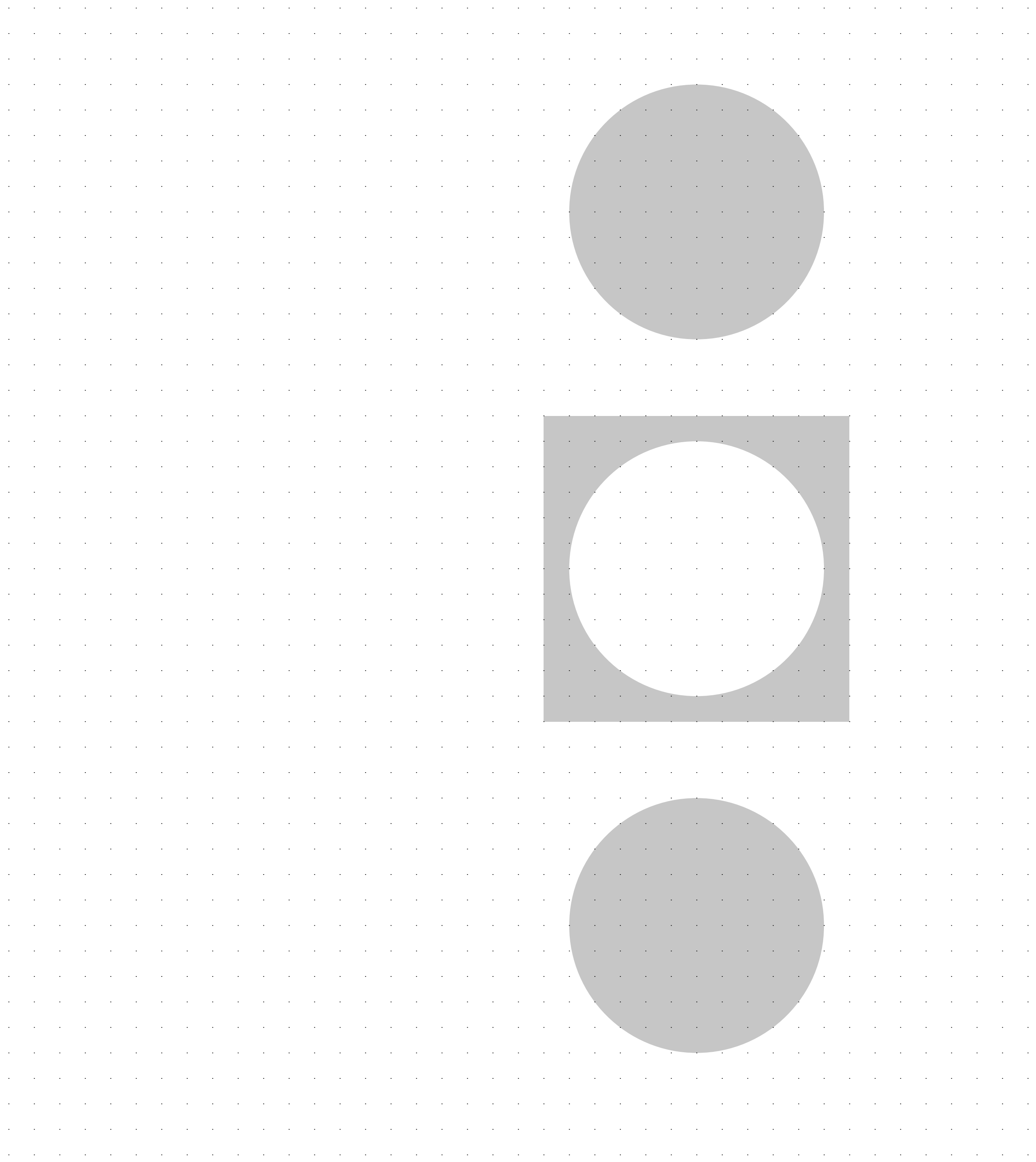
Console

When software runs, the computer performs operations at a rate too fast to perceive with human eyes. Because it is important to understand what is happening inside the machine, the functions `print()` and `println()` can be used to display data while a program is running. These functions don’t send pages to a printer, but instead write text to the console (pp. 8, 9). The console can be used to display a variable, confirm an event, or check incoming data from an external device. Such uses might not seem clear now, but they will reveal themselves over the course of this book. Like comments, `print()` and `println()` can clarify the intentions and execution of computer programs.

```
// To print text to the screen, place the desired output in quotes      1-07
println("Processing...");  // Prints "Processing..." to the console

// To print the value of a variable, rather than its name,
// don't put the name of the variable in quotes
int x = 20;
println(x);  // Prints "20" to the console

// While println() moves to the next line after the text
// is output, print() does not
print("10");
println("20");  // Prints "1020" to the console
println("30");  // Prints "30" to the console

// The "+" operator can be used for combining multiple text
// elements into one line
int x2 = 20;
int y2 = 80;
println(x2 + " : " + y2);  // Prints "20 : 80" to the message window
```

Exercises

1. *Write comments in the text area explaining a piece of software you would like to write.*
2. *Write a program to make a 640 × 480 pixel display window with a black background.*
3. *Use `print()` and `println()` to write some text to the console.*



Shape 1: Coordinates, Primitives

This unit introduces the coordinate system of the display window and a variety of geometric shapes.

Syntax introduced:
```
size(), point(), line(), triangle(), quad(), rect(), ellipse(), bezier()
background(), fill(), stroke(), noFill(), noStroke()
strokeWeight(), strokeCap(), strokeJoin()
smooth(), noSmooth(), ellipseMode(), rectMode()
```

Drawing a shape with code can be difficult because every aspect of its location must be specified with a number. When you're accustomed to drawing with a pencil or moving shapes around on a screen with a mouse, it can take time to start thinking in relation to the screen's strict coordinate grid. The mental gap between seeing a composition on paper or in your mind and translating it into code notation is wide, but easily bridged.

Coordinates

Before making a drawing, it's important to think about the dimensions and qualities of the surface to which you'll be drawing. If you're making a drawing on paper, you can choose from myriad utensils and papers. For quick sketching, newsprint and charcoal are appropriate. For a refined drawing, a smooth handmade paper and range of pencils may be preferred. In contrast, when you are drawing to a computer's screen, the primary options available are the size of the window and the background color.

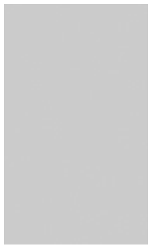
A computer screen is a grid of small light elements called pixels. Screens come in many sizes and resolutions. We have three different types of computer screens in our studios, and they all have a different number of pixels. The laptops have 1,764,000 pixels (1680 wide × 1050 high), the flat panels have 1,310,720 pixels (1280 wide × 1024 high), and the older monitors have 786,432 pixels (1024 wide × 768 high). Millions of pixels may sound like a vast quantity, but they produce a poor visual resolution compared to physical media such as paper. Contemporary screens have a resolution around 100 dots per inch, while many modern printers provide more than 1000 dots per inch. On the other hand, paper images are fixed, but screens have the advantage of being able to change their image many times per second.

Processing programs can control all or a subset of the screen's pixels. When you click the Run button, a display window opens and allows access to reading and writing the pixels within. It's possible to create images larger than the screen, but in most cases you'll make a window the size of the screen or smaller.

The size of the display window is controlled with the `size()` function:

```
size(width, height)
```

The `size()` function has two parameters: the first sets the width of the window and the second sets its height.

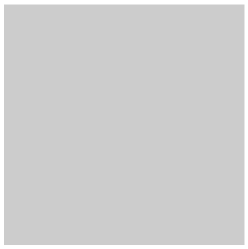


```
// Draw the display window 120 pixels
// wide and 200 pixels high
size(120, 200);
```

2-01

```
// Draw the display window 320 pixels
// wide and 240 pixels high
size(320, 240);
```

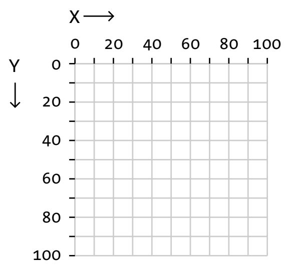
2-02

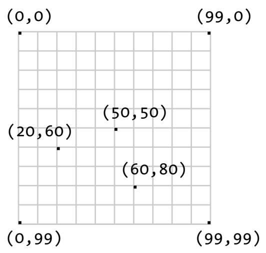


```
// Draw the display window 200 pixels
// wide and 200 pixels high
size(200, 200);
```

2-03

A position on the screen is comprised of an x-coordinate and a y-coordinate. The x-coordinate is the horizontal distance from the origin and the y-coordinate is the vertical distance. In Processing, the origin is the upper-left corner of the display window and coordinate values increase down and to the right. The image on the left shows the coordinate system, and the image on the right shows a few coordinates placed on the grid:





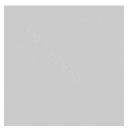
A position is written as the x-coordinate value followed by the y-coordinate, separated with a comma. The notation for the origin is (0,0), the coordinate (50,50) has an x-coordinate of 50 and a y-coordinate of 50, and the coordinate (20,60) is an x-coordinate of 20 and a y-coordinate of 60. If the size of the display window is 100 pixels wide and 100 pixels high, (0,0) is the pixel in the upper-left corner, (99,0) is the pixel in the upper-right corner, (0,99) is the pixel in the lower-left corner, and (99,99) is the pixel in the lower-right corner. This becomes clearer when we look at examples using `point()`.

Primitive shapes

A point is the simplest visual element and is drawn with the `point()` function:

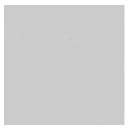
`point(x, y)`

This function has two parameters: the first is the x-coordinate and the second is the y-coordinate. Unless specified otherwise, a point is the size of a single pixel.



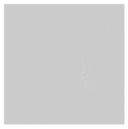
2-04

```
// Points with the same X and Y parameters
// form a diagonal line from the
// upper-left corner to the lower-right corner
point(20, 20);
point(30, 30);
point(40, 40);
point(50, 50);
point(60, 60);
```



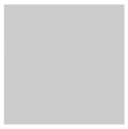
```
// Points with the same Y parameter have the
// same distance from the top and bottom
// edges of the frame
point(50, 30);
point(55, 30);
point(60, 30);
point(65, 30);
point(70, 30);
```

2-05



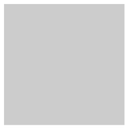
```
// Points with the same X parameter have the
// same distance from the left and right
// edges of the frame
point(70, 50);
point(70, 55);
point(70, 60);
point(70, 65);
point(70, 70);
```

2-06



```
// Placing a group of points next to one
// another creates a line
point(50, 50);
point(50, 51);
point(50, 52);
point(50, 53);
point(50, 54);
point(50, 55);
point(50, 56);
point(50, 57);
point(50, 58);
point(50, 59);
```

2-07



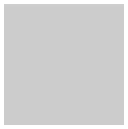
```
// Setting points outside the display
// area will not cause an error,
// but the points won't be visible
point(-500, 100);
point(400, -600);
point(140, 2500);
point(2500, 100);
```

2-08

While it's possible to draw any line as a series of points, lines are more simply drawn with the `line()` function. This function has four parameters, two for each endpoint:

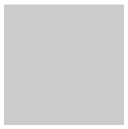
```
line(x1, y1, x2, y2)
```

The first two parameters set the position where the line starts and the last two set the position where the line stops.



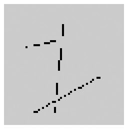
2-09

```
// When the y-coordinates for a line are the
// same, the line is horizontal
line(10, 30, 90, 30);
line(10, 40, 90, 40);
line(10, 50, 90, 50);
```



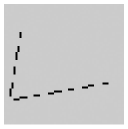
2-10

```
// When the x-coordinates for a line are the
// same, the line is vertical
line(40, 10, 40, 90);
line(50, 10, 50, 90);
line(60, 10, 60, 90);
```



2-11

```
// When all four parameters are different,
// the lines are diagonal
line(25, 90, 80, 60);
line(50, 12, 42, 90);
line(45, 30, 18, 36);
```



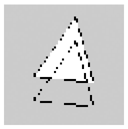
2-12

```
// When two lines share the same point they connect
line(15, 20, 5, 80);
line(90, 65, 5, 80);
```

The `triangle()` function draws triangles. It has six parameters, two for each point:

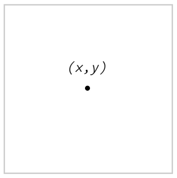
```
triangle(x1, y1, x2, y2, x3, y3)
```

The first pair defines the first point, the middle pair the second point, and the last pair the third point. Any triangle can be drawn by connecting three lines, but the `triangle()` function makes it possible to draw a filled shape. Triangles of all shapes and sizes can be created by changing the parameter values.

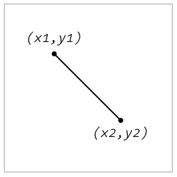


2-13

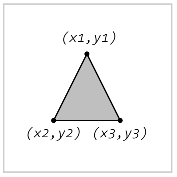
```
triangle(60, 10, 25, 60, 75, 65);  // Filled triangle
line(60, 30, 25, 80);  // Outlined triangle edge
line(25, 80, 75, 85);  // Outlined triangle edge
line(75, 85, 60, 30);  // Outlined triangle edge
```



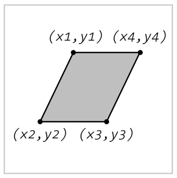
`point(x, y)`



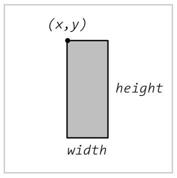
`line(x1, y1, x2, y2)`



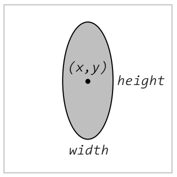
`triangle(x1, y1, x2, y2, x3, y3)`



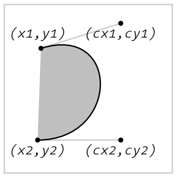
`quad(x1, y1, x2, y2, x3, y3, x4, y4)`



`rect(x, y, width, height)`

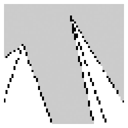


`ellipse(x, y, width, height)`



`bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)`

Geometry primitives
Processing has seven functions to assist in making simple shapes. These images show the format for each. Replace the parameters with numbers to use them within a program. These functions are demonstrated in codes 2-04 to 2-22.

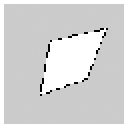


```
triangle(55, 9, 110, 100, 85, 100);
triangle(55, 9, 85, 100, 75, 100);
triangle(-1, 46, 16, 34, -7, 100);
triangle(16, 34, -7, 100, 40, 100);
```
2-14

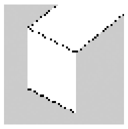
The `quad()` function draws a quadrilateral, a four-sided polygon. The function has eight parameters, two for each point.

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

Changing the parameter values can yield rectangles, squares, parallelograms, and irregular quadrilaterals.



```
quad(38, 31, 86, 20, 69, 63, 30, 76);
```
2-15

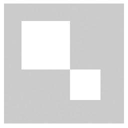


```
quad(20, 20, 20, 70, 60, 90, 60, 40);
quad(20, 20, 70, -20, 110, 0, 60, 40);
```
2-16

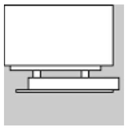
Drawing rectangles and ellipses works differently than the shapes previously introduced. Instead of defining each point, the four parameters set the position and the dimensions of the shape. The `rect()` function draws a rectangle:

```
rect(x, y, width, height)
```

The first two parameters set the location of the upper-left corner, the third sets the width, and the fourth sets the height. Use the same value for the *`width`* and *`height`* parameters to draw a square.



```
rect(15, 15, 40, 40);  // Large square
rect(55, 55, 25, 25);  // Small square
```
2-17

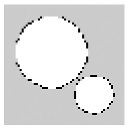


```
rect(0, 0, 90, 50);
rect(5, 50, 75, 4);
rect(24, 54, 6, 6);
rect(64, 54, 6, 6);
rect(20, 60, 75, 10);
rect(10, 70, 80, 2);
```
2-18

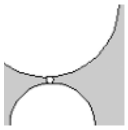
The `ellipse()` function draws an ellipse in the display window:

```
ellipse(x, y, width, height)
```

The first two parameters set the location of the center of the ellipse, the third sets the width, and the fourth sets the height. Use the same value for the *`width`* and *`height`* parameters to draw a circle.



2-19
```
ellipse(40, 40, 60, 60);  // Large circle
ellipse(75, 75, 32, 32);  // Small circle
```

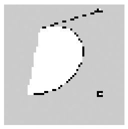


2-20
```
ellipse(35, 0, 120, 120);
ellipse(38, 62, 6, 6);
ellipse(40, 100, 70, 70);
```

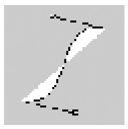
The `bezier()` function can draw lines that are not straight. A Bézier curve is defined by a series of control points and anchor points. A curve is drawn between the anchor points, and the control points define its shape:

```
bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)
```

The function requires eight parameters to set four points. The curve is drawn between the first and fourth points, and the control points are defined by the second and third points. In software that uses Bézier curves, such as Adobe Illustrator, the control points are represented by the tiny handles that protrude from the edge of a curve.



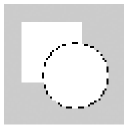
2-21
```
bezier(32, 20, 80, 5, 80, 75, 30, 75);
// Draw the control points
line(32, 20, 80, 5);
ellipse(80, 5, 4, 4);
line(80, 75, 30, 75);
ellipse(80, 75, 4, 4);
```



2-22
```
bezier(85, 20, 40, 10, 60, 90, 15, 80);
// Draw the control points
line(85, 20, 40, 10);
ellipse(40, 10, 4, 4);
line(60, 90, 15, 80);
ellipse(60, 90, 4, 4);
```

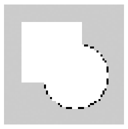
Drawing order

The order in which shapes are drawn in the code defines which shapes appear on top of others in the display window. If a rectangle is drawn in the first line of a program, it is drawn behind an ellipse drawn in the second line of the program. Reversing the order places the rectangle on top.



```
rect(15, 15, 50, 50);     // Bottom
ellipse(60, 60, 55, 55);  // Top
```

2-23

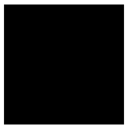


```
ellipse(60, 60, 55, 55);  // Bottom
rect(15, 15, 50, 50);     // Top
```

2-24

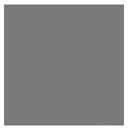
Gray values

The examples so far have used the default light-gray background, black lines, and white shapes. To change these default values, it's necessary to introduce additional syntax. The `background()` function sets the color of the display window with a number between 0 and 255. This range may be awkward if you're not familiar with drawing software on the computer. The value 255 is white and the value 0 is black, with a range of gray values in between. If no background value is defined, the default value 204 (light gray) is used.



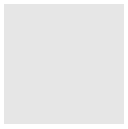
```
background(0);
```

2-25



```
background(124);
```

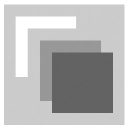
2-26



```
background(230);
```

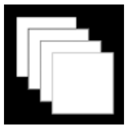
2-27

The `fill()` function sets the fill value of shapes, and the `stroke()` function sets the outline value of the drawn shapes. If no fill value is defined, the default value of 255 (white) is used. If no stroke value is defined, the default value of 0 (black) is used.



2-28

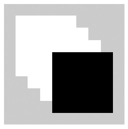
```
rect(10, 10, 50, 50);
fill(204);  // Light gray
rect(20, 20, 50, 50);
fill(153);  // Middle gray
rect(30, 30, 50, 50);
fill(102);  // Dark gray
rect(40, 40, 50, 50);
```



2-29

```
background(0);
rect(10, 10, 50, 50);
stroke(102); // Dark gray
rect(20, 20, 50, 50);
stroke(153); // Middle gray
rect(30, 30, 50, 50);
stroke(204); // Light gray
rect(40, 40, 50, 50);
```

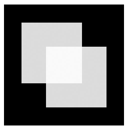
Once a fill or stroke value is defined, it applies to all shapes drawn afterward. To change the fill or stroke value, use the `fill()` or `stroke()` function again.



2-30

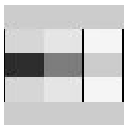
```
fill(255);  // White
rect(10, 10, 50, 50);
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
fill(0);  // Black
rect(40, 40, 50, 50);
```

An optional second parameter to `fill()` and `stroke()` controls transparency. Setting the parameter to 255 makes the shape entirely opaque, and 0 is totally transparent:



2-31

```
background(0);
fill(255, 220);
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```

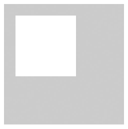


2-32

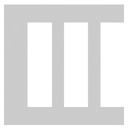
```
fill(0);
rect(0, 40, 100, 20);
fill(255, 51);   // Low opacity
rect(0, 20, 33, 60);
fill(255, 127);  // Medium opacity
```

```
rect(33, 20, 33, 60);
fill(255, 204);  // High opacity
rect(66, 20, 33, 60);
```
2-32 cont.

The stroke and fill of a shape can be disabled. The `noFill()` function stops Processing from filling shapes, and the `noStroke()` function stops lines from being drawn and shapes from having outlines. If `noFill()` and `noStroke()` are both used, nothing will be drawn to the screen.



```
rect(10, 10, 50, 50);
noFill();  // Disable the fill
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
```
2-33

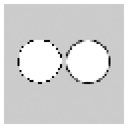


```
rect(20, 15, 20, 70);
noStroke();  // Disable the stroke
rect(50, 15, 20, 70);
rect(80, 15, 20, 70);
```
2-34

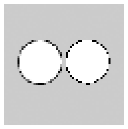
Setting color fill and stroke values is introduced in Color 1 (p. 85).

Drawing attributes

In addition to changing the fill and stroke values of shapes, it's also possible to change attributes of the geometry. The `smooth()` and `noSmooth()` functions enable and disable smoothing (also called antialiasing). Once these functions are used, all shapes drawn afterward are affected. If `smooth()` is used first, using `noSmooth()` cancels the setting, and vice versa.



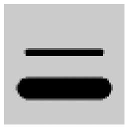
```
ellipse(30, 48, 36, 36);
smooth();
ellipse(70, 48, 36, 36);
```
2-35



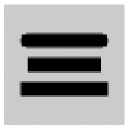
```
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```
2-36

Line attributes are controlled by the `strokeWeight()`, `strokeCap()`, and `strokeJoin()` functions. The `strokeWeight()` function has one numeric parameter that sets the thickness of all lines drawn after the function is used. The `strokeCap()` function requires one parameter that can be either `ROUND`, `SQUARE`, or `PROJECT`.

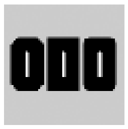
ROUND makes round endpoints, and SQUARE squares them. PROJECT is a mix of the two that extends a SQUARE endpoint by the radius of the line. The `strokeJoin()` function has one parameter that can be either BEVEL, MITER, or ROUND. These parameters determine the way line segments or the stroke around a shape connects. BEVEL causes lines to join with squared corners, MITER is the default and joins lines with pointed corners, and ROUND creates a curve.



2-37
```
smooth();
line(20, 20, 80, 20);  // Default line weight of 1
strokeWeight(6);
line(20, 40, 80, 40);  // Thicker line
strokeWeight(18);
line(20, 70, 80, 70);  // Beastly line
```



2-38
```
smooth();
strokeWeight(12);
strokeCap(ROUND);
line(20, 30, 80, 30);  // Top line
strokeCap(SQUARE);
line(20, 50, 80, 50);  // Middle line
strokeCap(PROJECT);
line(20, 70, 80, 70);  // Bottom line
```

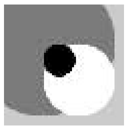


2-39
```
smooth();
strokeWeight(12);
strokeJoin(BEVEL);
rect(12, 33, 15, 33);  // Left shape
strokeJoin(MITER);
rect(42, 33, 15, 33);  // Middle shape
strokeJoin(ROUND);
rect(72, 33, 15, 33);  // Right shape
```

Shape 2 (p. 69) and Shape 3 (p. 197) show how to draw shapes with more flexibility.

Drawing modes

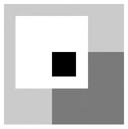
By default, the parameters for `ellipse()` set the x-coordinate of the center, the y-coordinate of the center, the width, and the height. The `ellipseMode()` function changes the way these parameters are used to draw ellipses. The `ellipseMode()` function requires one parameter that can be either CENTER, RADIUS, CORNER, or CORNERS. The default mode is CENTER. The RADIUS mode also uses the first and second parameters of `ellipse()` to set the center, but causes the third parameter to set half of the width and the fourth parameter to set half of the height. The CORNER mode makes ellipse() work similarly to rect(). It causes the first and second parameters to position the upper-left corner of the rectangle that circumscribes the ellipse and uses the third and fourth parameters to set the width and height. The CORNERS mode has a similar affect to CORNER, but is causes the third and fourth parameters to ellipse() to set the lower-right corner of the rectangle.



2-40

```
smooth();
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60);  // Gray ellipse
fill(255);
ellipseMode(CORNER);
ellipse(33, 33, 60, 60);  // White ellipse
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60);  // Black ellipse
```

In a similar fashion, the rectMode() function affects how rectangles are drawn. It requires one parameter that can be either CORNER, CORNERS, or CENTER. The default mode is CORNER, and CORNERS causes the third and fourth parameters of rect() to draw the corner opposite the first. The CENTER mode causes the first and second parameters of rect() to set the center of the rectangle and uses the third and fourth parameters as the width and height.



2-41

```
noStroke();
rectMode(CORNER);
fill(126);
rect(40, 40, 60, 60);     // Gray ellipse
rectMode(CENTER);
fill(255);
rect(40, 40, 60, 60);     // White ellipse
rectMode(CORNERS);
fill(0);
rect(40, 40, 60, 60);     // Black ellipse
```

Exercises

1. *Create a composition by carefully positioning one line and one ellipse.*
2. *Modify the code for exercise 1 to change the fill, stroke, and background values.*
3. *Create a visual knot using only Bézier curves.*

false	84	6	456749	590203.000
true	12	*	418953	1209181.500
true	-64	D	485484	2082378.000
false	96	"	1895150	-1391668.750
true	-48	,	-567039	-1517834.000
false	-123	x	200745	-2107862.750
false	15	)	-1061358	537499.250
true	-23	!	1186751	2047401.000
false	47	M	1579167	1057598.500
false	59	U	1362780	-2092479.125
false	-23	Q	969695	1715130.000
false	-94	S	-1790733	-1348664.000
false	-50	c	734766	923550.750
true	-92	A	-159095	-969601.250
true	42	&	813913	-651030.125
false	-41	Y	1543424	1857006.000
false	62	q	145359	483532.000
true	-83	=	-1213025	-1273871.000
false	-107	%	-366036	825891.750
true	69	y	913485	815780.500
true	-116	i	-450082	1912691.000
false	-125	Z	-454392	479536.000
false	-122	i	1846579	2119445.000
true	97	M	-1416086	-1530103.000
false	-80	d	1736539	761730.000
true	77	]	-251570	-770656.500
true	-104	p	-122459	7880.500
false	125	U	2015498	281250.000
true	-32	$	1235768	-1839862.000
true	-121	*	-70136	-2110472.000
false	-5	|	-644916	1148654.500
false	0	\	-1050010	-697901.375
true	61	o	973362	-1923781.750
true	-126	2	-627696	1375153.750
true	108	o	1440987	-2131691.000
false	-49	c	135757	-848097.250
false	-48	:	1884981	1607443.250
false	27	S	1755091	-1509226.500
true	122	\	-771512	-13727.750
true	1	N	88857	-1286271.875
true	-31	L	225379	876042.750
true	-46	9	-2097426	1962124.500
false	66	f	1030910	460799.250
true	-68	U	-476555	-1738484.250
false	-42	/	347440	-2130674.500
false	-60	%	1687135	1093877.250
true	-72	H	-548337	805527.750
true	-24	_	-631134	-1417360.000
false	-39	e	1491429	1694085.750
false	20	q	46194	-1970949.250
true	37	R	486794	2031981.250
true	28	F	-2030792	1623354.000
false	44	[	-855542	1365101.250
false	61	=	564487	-372181.625

Data 1: Variables

This unit introduces different types of data and explains how to create variables and assign them values.

Syntax introduced:
`int, float, boolean, true, false, = (assign), width, height`

What is data? Data often consists of measurements of physical characteristics. For example, Casey's California driver's license states his sex is M, his hair is BRN, and his eyes are HZL. The values M, BRN, and HZL are items of data associated with Casey. Data can be the population of a country, the average annual rainfall in Los Angeles, or your current heart rate. In software, data is stored as numbers and characters. Examples of digital data include a photograph of a friend stored on your hard drive, a song downloaded from the Internet, and a news article loaded through a web browser. Less obvious is the data continually created and exchanged between computers and other devices. For example, computers are continually receiving data from the mouse and keyboard. When writing a program, you might create a data element to save the location of a shape, to store a color for later use, or to continuously measure changes in cursor position.

Data types

Processing can store and modify many different kinds of data, including numbers, letters, words, colors, images, fonts, and boolean values (`true`, `false`). The computer stores each in a different way, so it has to know which type of data is being used to know how to manage it. For example, storing a word takes more room than storing one letter; therefore, storing the word *Cincinnati* requires more space than storing the letter *C*. If space has been allocated for only one letter, trying to store a word in the same space will cause an error. Every data element is represented as sequences of bits (0s and 1s) in the computer's memory (more information about bits is found in Appendix D, p. 669). For example, 01000001 can be interpreted as the letter *A*, and it can also be interpreted as the number 65. It's necessary to specify the type of data so the computer knows how to correctly interpret the bits.

Numeric data is the first type of data encountered in the following sections of this book. There are two types of numeric data used in Processing: integer and floating-point. Integers are whole numbers such as 12, –120, 8, and 934. Processing represents integer data with the `int` data type. Floating-point numbers have a decimal point for creating fractions of whole numbers such as 12.8, –120.75, 8.125, and 934.82736. Processing represents floating-point data with the `float` data type. Floating-point numbers are often used to approximate analog or continuous values because they have decimal resolution. For example, using integer values, there is only one number between 3 and 5, but floating-point numbers allow us to express myriad numbers between such as 4.0, 4.5, 4.75, 4.825, etc. Both int and float values may be positive, negative, or zero.

The simplest data element in Processing is a boolean variable. Variables of this type can have only one of two values—`true` or `false`. The name boolean refers to the mathematician George Boole (b. 1815), the inventor of Boolean algebra—the foundation for how digital computers work. A boolean variable is often used to make decisions about which lines of code are run and which are ignored.

The following table compares the capacities of the data types mentioned above with other common data types:

Name	Size	Value range
boolean	1 bit	true or false
byte	8 bits	-128 to 127
char	16 bits	0 to 65535
int	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.40282347E+38 to -3.40282347E+38
color	32 bits	16,777,216 colors

Additional types of data are introduced and explained in Data 2 (p. 101), Data 3 (p. 105), Image 1 (p. 95), Typography 1 (p. 111), and Structure 4 (p. 395).

Variables

A variable is a container for storing data. Variables allow a data element to be reused many times within a program. Every variable has two parts, a name and a value. If the number 21 is stored in the variable called *age*, every time the word *age* appears in the program, it will be replaced with the value 21 when the code is run. In addition to its name and value, every variable has a data type that defines the category of data it can hold.

A variable must be declared before it is used. A variable declaration states the data type and variable name. The following lines declare variables and then assign values to the variables:

3-01
```
int x;      // Declare the variable x of type int
float y;    // Declare the variable y of type float
boolean b;  // Declare the variable b of type boolean
x = 50;     // Assign the value 50 to x
y = 12.6;   // Assign the value 12.6 to y
b = true;   // Assign the value true to b
```

As a shortcut, a variable can be declared and assigned on the same line:

```
int x = 50;                                                     3-02
float y = 12.6;
boolean b = true;
```

More than one variable can be declared in one line, and the variables can then be assigned separately:

```
float x, y, z;                                                  3-03
x = -3.9;
y = 10.1;
z = 124.23;
```

When a variable is declared, it is necessary to state the data type before its name; but after it's declared, the data type cannot be changed or restated. If the data type is included again for the same variable, the computer will interpret this as an attempt to make a new variable with the same name, and this will cause an error (an exception to this rule is made when each variable has a different scope, p. 178):

```
int x = 69;  // Assign 69 to x                                  3-04
x = 70;      // Assign 70 to x
int x = 71;  // ERROR! The data type for x is duplicated
```

The = symbol is called the assignment operator. It assigns the value from the right side of the = to the variable on its left. Values can be assigned only to variables. Trying to assign a constant to another constant produces an error:

```
// Error! The left side of an assignment must be a variable     3-05
5 = 12;
```

When working with variables of different types in the same program, be careful not to mix types in a way that causes an error. For example, it's not possible to fit a floating-point number into an integer variable:

```
// Error! It's not possible to fit a floating-point number into an int   3-06
int x = 24.8;
```

```
float f = 12.5;                                                          3-07
// Error! It's not possible to fit a floating-point number into an int
int y = f;
```

Variables should have names that describe their content. This makes programs easier to read and can reduce the need for verbose commenting. It's up to the programmer to decide how she will name variables. For example, a variable storing the room temperature could logically have the following names:

```
t
temp
temperature
roomTemp
roomTemperature
```

Variables like `t` should be used minimally or not at all because they are cryptic—there's no hint as to what they contain. However, long names such as *`roomTemperature`* can also make code tedious to read. If we were writing a program with this variable, our preference might be to use the name *`roomTemp`* because it is both concise and descriptive. The name *`temp`* could also work, but because it's used commonly as an abbreviation for "temporary," it wouldn't be the best choice.

There are a few conventions that make it easier for other people to read your programs. Variables' names should start with a lowercase letter, and if there are multiple words in the name, the first letter of each additional word should be capitalized. There are a few absolute rules in naming variables. Variable names cannot start with numbers, and they must not be a reserved word. Examples of reserved words include `int`, `if`, `true`, and `null`. A complete list is found in Appendix B (p. 663). To avoid confusion, variables should not have the same names as elements of the Processing language such as `line` and `ellipse`. The complete Processing language is listed in the reference included with the software.

Another important consideration related to variables is the scope (p. 178). The scope of a variable defines where it can be used relative to where it's created.

Processing variables

The Processing language has built-in variables for storing commonly used data. The width and height of the display window are stored in variables called `width` and `height`. If a program doesn't include `size()`, the `width` and `height` variables are both set to 100. Test by running the following programs

```
println(width + ", " + height);  // Prints "100, 100" to the console     3-08
```

```
size(300, 400);                                                          3-09
println(width + ", " + height);  // Prints "300, 400" to the console
```

```
size(1280, 1024);                                                        3-10
println(width + ", " + height);  // Prints "1280, 1024" to the console
```

Using the `width` and `height` variables is useful when writing a program to scale to different sizes. This technique allows a simple change to the parameters of `size()` to alter the dimensions and proportions of a program, rather than changing values throughout the code. Run the following code with different values in the `size()` function to see it scale to every window size.

3-11

```
size(100, 100);
ellipse(width*0.5, height*0.5, width*0.66, height*0.66);
line(width*0.5, 0, width*0.5, height);
line(0, height*0.5, width, height*0.5);
```

You should always use actual numbers in `size()` instead of variables. When a sketch is exported, these numbers are used to determine the dimension of the sketch on its Web page. More information about this can be seen in the reference for `size()`.

Processing variables that store the cursor position and the most recent key pressed are discussed in Input 1 (p. 205) and Input 2 (p. 223).

Exercises

1. *Think about different types of numbers you use daily. Are they integer or floating-point numbers?*
2. *Make a few `int` and `float` variables. Try assigning them in different ways. Write the values to the console with `println()`.*
3. *Create a composition that scales proportionally with different window sizes. Put different values into `size()` to test.*