

Data 4: Arrays

This unit introduces arrays of data.

Syntax introduced:

Array, [] (array access), new, Array.length
append(), shorten(), expand(), arraycopy()

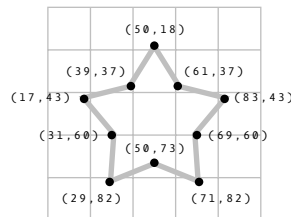
The term array refers to a structured grouping or an imposing number—“The dinner buffet offers an array of choices,” “The city of Los Angeles faces an array of problems.” In computer programming, an array is a set of data elements stored under the same name. Arrays can be created to hold any type of data, and each element can be individually assigned and read. There can be arrays of numbers, characters, sentences, boolean values, etc. Arrays might store vertex data for complex shapes, recent keystrokes from the keyboard, or data read from a file.

Five integer variables (1919, 1940, 1975, 1976, 1990) can be stored in one integer array rather than defining five separate variables. For example, let’s call this array “dates” and store the values in sequence:

dates	1919	1940	1975	1976	1990
	[0]	[1]	[2]	[3]	[4]

Array elements are numbered starting with zero, which may seem confusing at first but is important for more advanced programming. The first element is at position [0], the second is at [1], and so on. The position of each element is determined by its offset from the start of the array. The first element is at position [0] because it has no offset; the second element is at position [1] because it is offset one place from the beginning. The last position in the array is calculated by subtracting 1 from the array length. In this example, the last element is at position [4] because there are five elements in the array.

Arrays can make the task of programming much easier. While it’s not necessary to use them, they can be valuable structures for managing data. Let’s begin with a set of data points we want to add to our program in order to draw a star:



The following example to draw this shape demonstrates some of the benefits of using arrays, like avoiding the cumbersome chore of storing data points in individual

variables. The star has 10 vertex points, each with 2 values, for a total of 20 data elements. Inputting this data into a program requires either creating 20 variables or using an array. The code (below) on the left demonstrates using separate variables. The code in the middle uses 10 arrays, one for each point of the shape. This use of arrays improves the situation, but we can do better. The code on the right shows how the data elements can be logically grouped together in two arrays, one for the x-coordinates and one for the y-coordinates.

Separate variables

```
int x0 = 50;
int y0 = 18;
int x1 = 61;
int y1 = 37;
int x2 = 83;
int y2 = 43;
int x3 = 69;
int y3 = 60;
int x4 = 71;
int y4 = 82;
int x5 = 50;
int y5 = 73;
int x6 = 29;
int y6 = 82;
int x7 = 31;
int y7 = 60;
int x8 = 17;
int y8 = 43;
int x9 = 39;
int y9 = 37;
```

One array for each point

```
int[] p0 = { 50, 18 };
int[] p1 = { 61, 37 };
int[] p2 = { 83, 43 };
int[] p3 = { 69, 60 };
int[] p4 = { 71, 82 };
int[] p5 = { 50, 73 };
int[] p6 = { 29, 82 };
int[] p7 = { 31, 60 };
int[] p8 = { 17, 43 };
int[] p9 = { 39, 37 };
```

One array for each axis

```
int[] x = { 50, 61, 83, 69, 71,
           50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82,
           73, 82, 60, 43, 37 };
```

This example shows how to use the arrays within a program. The data for each vertex is accessed in sequence with a `for` structure. The syntax and usage of arrays is discussed in more detail in the following pages.



```
int[] x = { 50, 61, 83, 69, 71, 50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82, 73, 82, 60, 43, 37 };
```

33-01

```
beginShape();
// Reads one array element every time through the for()
for (int i = 0; i < x.length; i++) {
    vertex(x[i], y[i]);
}
endShape(CLOSE);
```

Using arrays

Arrays are declared similarly to other data types, but they are distinguished with brackets, [and]. When an array is declared, the type of data it stores must be specified. After the array is declared, the array must be created with the keyword “new.” This additional step allocates space in the computer’s memory to store the array’s data. After the array is created, the values can be assigned. There are different ways to declare, create, and assign arrays. In the following examples explaining these differences, an array with five elements is created and filled with the values 19, 40, 75, 76, and 90. Note the different way each method for creating and assigning elements of the array relates to `setup()`.

```
int[] data; // Declare 33-02
void setup() {
    size(100, 100);
    data = new int[5]; // Create
    data[0] = 19; // Assign
    data[1] = 40;
    data[2] = 75;
    data[3] = 76;
    data[4] = 90;
}
```

```
int[] data = new int[5]; // Declare, create 33-03
void setup() {
    size(100, 100);
    data[0] = 19; // Assign
    data[1] = 40;
    data[2] = 75;
    data[3] = 76;
    data[4] = 90;
}
```

```
int[] data = { 19, 40, 75, 76, 90 }; // Declare, create, assign 33-04
void setup() {
    size(100, 100);
}
```


The previous three examples assume the arrays are used in a sketch with `setup()` and `draw()`. If arrays are not used with these functions, they can be created and assigned in the simpler ways shown in the following examples.

```
int[] data;           // Declare                                     33-05
data = new int[5];   // Create
data[0] = 19;        // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```

```
int[] data = new int[5]; // Declare, create                       33-06
data[0] = 19;           // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```

```
int[] data = { 19, 40, 75, 76, 90 }; // Declare, create, assign 33-07
```

The declare, create, and assign steps allow an array's values to be read. An array element is accessed using the name of the variable followed by brackets around the position from which you are trying to read.

```
 int[] data = { 19, 40, 75, 76, 90 };                               33-08
line(data[0], 0, data[0], 100);
line(data[1], 0, data[1], 100);
line(data[2], 0, data[2], 100);
line(data[3], 0, data[3], 100);
line(data[4], 0, data[4], 100);
```

Remember, the first element in the array is in the 0 position. If you try to access a member of the array that lies outside the array boundaries, your program will terminate and give an `ArrayIndexOutOfBoundsException`.

```
int[] data = { 19, 40, 75, 76, 90 };                               33-09
println(data[0]); // Prints 19 to the console
println(data[2]); // Prints 75 to the console
println(data[5]); // ERROR! The last element of the array is 4
```

The `length` field stores the number of elements in an array. This field is stored within the array and can be accessed with the dot operator (p. 107–108). The following example demonstrates how to utilize it.

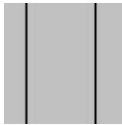
```

int[] data1 = { 19, 40, 75, 76, 90 };
int[] data2 = { 19, 40 };
int[] data3 = new int[127];
println(data1.length); // Prints "5" to the console
println(data2.length); // Prints "2" to the console
println(data3.length); // Prints "127" to the console

```

33-10

Usually, a `for` structure is used to access array elements, especially with large arrays. The following example draws the same lines as code 33-08 but uses a `for` structure to iterate through every value in the array.



```

int[] data = { 19, 40, 75, 76, 90 };
for (int i = 0; i < data.length; i++) {
    line(data[i], 0, data[i], 100);
}

```

33-11

A `for` structure can also be used to put data inside an array—for instance, it can calculate a series of numbers and then assign each value to an array element. The following example stores the values from the `sin()` function in an array within `setup()` and then displays these values as the stroke values for lines within `draw()`.



```

float[] sineWave = new float[width];

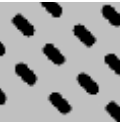
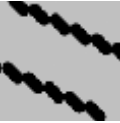
for (int i = 0; i < width; i++) {
    // Fill the array with values from sin()
    float r = map(i, 0, width, 0, TWO_PI);
    sineWave[i] = abs(sin(r));
}

for (int i = 0; i < sineWave.length; i++) {
    // Set the stroke values to numbers read from the array
    stroke(sineWave[i] * 255);
    line(i, 0, i, height);
}

```

33-12

Storing the coordinates of many elements is another way to use arrays to make a program easier to read and manage. In the following example, the `x[]` array stores the x-coordinate for each of the 12 elements in the array, and the `speed[]` array stores a rate corresponding to each. Writing this program without arrays would have required 24 separate variables. Instead, it's written in a flexible way; simply changing the value assigned to `numLines` sets the number of elements drawn to the screen.



```

int numLines = 12;
float[] x = new float[numLines];
float[] speed = new float[numLines];
float offset = 8; // Set space between lines

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(10);
  for (int i = 0; i < numLines; i++) {
    x[i] = i; // Set initial position
    speed[i] = 0.1 + (i / offset); // Set initial speed
  }
}

void draw() {
  background(204);
  for (int i = 0; i < x.length; i++) {
    x[i] += speed[i]; // Update line position
    if (x[i] > (width + offset)) { // If off the right,
      x[i] = -offset * 2; // return to the left
    }
    float y = i * offset; // Set y-coordinate for line
    line(x[i], y, x[i]+offset, y+offset); // Draw line
  }
}

```

Storing mouse data

Arrays are often used to store data from the mouse. The `pmouseX` and `pmouseY` variables store the cursor coordinates from the previous frame, but there's no built-in way to access the cursor values from earlier frames. At every frame, the `mouseX`, `mouseY`, `pmouseX`, and `pmouseY` variables are replaced with new numbers and their previous numbers are discarded. Creating an array is the easiest way to store the history of these values. In the following example, the most recent 100 values from `mouseY` are stored in an array and displayed on screen as a line from the left to the right edge of the screen. At each frame, the values in the array are shifted to the right and the newest value is added to the beginning.



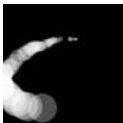
```
int[] y;

void setup() {
  size(100, 100);
  y = new int[width];
}

void draw() {
  background(204);
  // Shift the values to the right
  for (int i = y.length-1; i > 0; i--) {
    y[i] = y[i-1];
  }
  // Add new values to the beginning
  y[0] = constrain(mouseY, 0, height-1);
  // Display each pair of values as a line
  for (int i = 1; i < y.length; i++) {
    line(i, y[i], i-1, y[i-1]);
  }
}
```

33-14

Apply the same code simultaneously to the mouseX and mouseY values to store the position of the cursor. Displaying these values each frame creates a trail behind the cursor.



```
int num = 50;
int[] x = new int[num];
int[] y = new int[num];

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  fill(255, 102);
}

void draw() {
  background(0);
  // Shift the values to the right
  for (int i = num-1; i > 0; i--) {
    x[i] = x[i-1];
    y[i] = y[i-1];
  }
}
```

33-15

```

        // Add the new values to the beginning of the array
        x[0] = mouseX;
        y[0] = mouseY;
        // Draw the circles
        for (int i = 0; i < num; i++) {
            ellipse(x[i], y[i], i/2.0, i/2.0);
        }
    }
}

```

33-15
cont.

The following example produces the same result as the previous one but uses a more efficient technique. Instead of sorting the array elements in each frame, the program writes the new data to the next available array position. The elements in the array remain in the same position once they are written, but they are read in a different order each frame. Reading begins at the location of the oldest data element and continues to the end of the array. At the end of the array, the % operator (p. 45) is used to wrap back to the beginning. This technique is especially useful with larger arrays, to avoid unnecessary copying of data that can slow down a program.

```

int num = 50;
int[] x = new int[num];
int[] y = new int[num];
int indexPosition = 0;

```

33-16

```

void setup() {
    size(100, 100);
    noStroke();
    smooth();
    fill(255, 102);
}

void draw() {
    background(0);
    x[indexPosition] = mouseX;
    y[indexPosition] = mouseY;
    // Cycle between 0 and the number of elements
    indexPosition = (indexPosition + 1) % num;
    for (int i = 0; i < num; i++) {
        // Set the array position to read
        int pos = (indexPosition + i) % num;
        float radius = (num-i) / 2.0;
        ellipse(x[pos], y[pos], radius, radius);
    }
}
}

```


Array functions

Processing provides a group of functions that assist in managing array data. Only four of these functions are introduced here, but more are explained in the Extended Reference included with the software and available online at www.processing.org/reference.

The `append()` function expands an array by one element, adds data to the new position, and returns the new array:

```
String[] trees = { "ash", "oak" };                                     33-17
append(trees, "maple"); // INCORRECT! Does not change the array
print(trees); // Prints "ash oak"
println();
trees = append(trees, "maple"); // Add "maple" to the end
print(trees); // Prints "ash oak maple"
println();
// Add "beech" to the end of the trees array, and creates a new
// array to store the change
String[] moretrees = append(trees, "beech");
print(moretrees); // Prints "ash oak maple beech"
```

The `shorten()` function decreases an array by one element by removing the last element and returns the shortened array:

```
String[] trees = { "lychee", "coconut", "fig"};                       33-18
trees = shorten(trees); // Remove the last element from the array
print(trees); // Prints "lychee coconut"
println();
trees = shorten(trees); // Remove the last element from the array
print(trees); // Prints "lychee"
```

The `expand()` function increases the size of an array. It can expand to a specific size, or if no size is specified, the array's length will be doubled. If an array needs to have many additional elements, it's faster to use `expand()` to double the size than to use `append()` to continually add one value. The following example saves a new `mouseX` value to an array every frame. When the array becomes full, the size of the array is doubled and new `mouseX` values proceed to fill the enlarged array.

```
int[] x = new int[100]; // Array to store x-coordinates                33-19
int count; // Store the number of array positions

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

```

void draw() {
    x[count] = mouseX;           // Assign new x-coordinate to the array
    count++;                     // Increment the counter
    if (count == x.length) {    // If the x array is full,
        x = expand(x);          // double the size of x
        println(x.length);     // Write the new size to the console
    }
}

```

33-19
cont.

Array values cannot be copied with the assignment operator because they are objects. The most common way to copy elements from one array to another is to use special functions or to copy each element individually within a `for` structure. The `arraycopy()` function is the most efficient way to copy the entire contents of one array to another. The data is copied from the array used as the first parameter to the array used as the second parameter. Both arrays must be the same length for it to work in the configuration shown below.

```

String[] north = { "OH", "IN", "MI" };
String[] south = { "GA", "FL", "NC" };
arraycopy(north, south); // Copy from north array to south array
print(south); // Prints "OH IN MI"
println();
String[] east = { "MA", "NY", "RI" };
String[] west = new String[east.length]; // Create a new array
arraycopy(east, west); // Copy from east array to west array
print(west); // Prints "MA NY RI"

```

33-20

New functions can be written to perform operations on arrays, but arrays behave differently than data types such as `int` and `char`. When an array is used as a parameter to a function, the address (location in memory) of the array is transferred into the function instead of the actual data. No new array is created, and changes made within the function affect the array used as the parameter.

In the following example, the `data[]` array is used as the parameter to `halve()`. The address of `data[]` is passed to the `d[]` array in the `halve()` function. Because the address of `d[]` and `data[]` is the same, they both affect the same data. When changes are made to `d[]` on line 14, these changes are made to the values in `data[]`. The `draw()` function is not used because the calculation is made only once and nothing is drawn to the display window.

```
float[] data = { 19.0, 40.0, 75.0, 76.0, 90.0 };
```

33-21

```
void setup() {
    halve(data);
    println(data[0]); // Prints "9.5"
    println(data[1]); // Prints "20.0"
    println(data[2]); // Prints "37.5"
    println(data[3]); // Prints "38.0"
    println(data[4]); // Prints "45.0"
}

void halve(float[] d) {
    for (int i = 0; i < d.length; i++) { // For each array element,
        d[i] = d[i] / 2.0; // divide the value by 2
    }
}
```

Changing array data within a function without modifying the original array requires some additional lines of code. In the following example, the array is passed into the function as a parameter, a new array is made, the values from the original array are copied in the new array, changes are made to the new array, and finally the modified array is returned. Like the previous example, the `draw()` function is not used because nothing is drawn to the display window and the calculation is made only once.

```
float[] data = { 19.0, 40.0, 75.0, 76.0, 90.0 };
float[] halfData;
```

33-22

```
void setup() {
    halfData = halve(data); // Run the halve() function
    println(data[0] + ", " + halfData[0]); // Prints "19.0, 9.5"
    println(data[1] + ", " + halfData[1]); // Prints "40.0, 20.0"
    println(data[2] + ", " + halfData[2]); // Prints "75.0, 37.5"
    println(data[3] + ", " + halfData[3]); // Prints "76.0, 38.0"
    println(data[4] + ", " + halfData[4]); // Prints "90.0, 45.0"
}

float[] halve(float[] d) {
    float[] numbers = new float[d.length]; // Create a new array
    arraycopy(d, numbers);
    for (int i = 0; i < numbers.length; i++) { // For each element,
        numbers[i] = numbers[i] / 2; // divide the value by 2
    }
    return numbers; // Return the new array
}
```

Two-dimensional arrays

Data can also be stored and retrieved from arrays with more than one dimension. Using the example from the beginning of this unit, the data points for the star are put into a 2D array instead of two 1D arrays:

points	[0]	50	61	83	69	71	50	29	31	17	39
	[1]	18	37	43	60	82	73	82	60	43	37
		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

A 2D array is essentially a list of 1D arrays. It must be declared, then created, and then the values can be assigned just as in a 1D array. The following syntax converts this array to code:

```
int[][] points = { {50,18}, {61,37}, {83,43}, {69,60}, {71,82},  
                  {50,73}, {29,82}, {31,60}, {17,43}, {39,37} }; 33-23
```

```
println(points[4][0]); // Prints 71  
println(points[4][1]); // Prints 82  
println(points[4][2]); // ERROR! This element is outside the array  
println(points[0][0]); // Prints 50  
println(points[9][1]); // Prints 37
```

This program shows how it all fits together.



```
int[][] points = { {50,18}, {61,37}, {83,43}, {69,60},  
                  {71,82}, {50,73}, {29,82}, {31,60},  
                  {17,43}, {39,37} }; 33-24
```



```
void setup() {  
  size(100, 100);  
  fill(0);  
  smooth();  
}
```



```
void draw() {  
  background(204);  
  translate(mouseX - 50, mouseY - 50);  
  beginShape();  
  for (int i = 0; i < points.length; i++) {  
    vertex(points[i][0], points[i][1]);  
  }  
  endShape();  
}
```

It's possible to continue and make 3D and 4D arrays by extrapolating these techniques. However, multidimensional arrays can be confusing, and it's often a better idea to maintain multiple 1D or 2D arrays.

Exercises

1. *Create an array to store the y-coordinates of a sequence of shapes. Draw each shape inside `draw()` and use the values from the array to set the y-coordinate of each.*
2. *Write a function to multiply the values from two arrays together and return the result as a new array. Print the results to the console.*
3. *Use a 2D array to store the coordinates for a shape of your own invention. Use a `for` structure to draw the shape to the display window.*



Image 2: Animation

This unit introduces techniques for displaying sequences of images successively, creating animation.

Animation occurs when a series of images, each slightly different, are presented in quick succession. A diverse medium with a century of history, animation has progressed from the initial experiments of Winsor McCay to the commercial and realistic innovations of early Walt Disney studio productions, to the experimental films by such animators as Lotte Reiniger and James Whitney in the mid-twentieth century. The high volume of animated special effects in live-action film and the deluge of animated children's films are changing the role of animation in popular culture.

There's a long history of using software to extend the boundaries of animation. Some of the first computer graphics were presented as animation on film during the 1960s. Because of the cost and expertise required to make these films, they emerged from high-profile research facilities such as Bell Laboratories and IBM's Scientific Center. Kenneth C. Knowlton, then a researcher at Bell Labs, is an important protagonist in the story of early computer animation. He worked separately with artists Stan VanDerBeek and Lillian Schwartz to produce some of the first films made using computer graphics. VanDerBeek and Knowlton's *Poem Field* films, produced throughout the 1960s, utilized Knowlton's BEFLIX code and punch cards to produce permutations of visual micropatterns. Schwartz and Knowlton's *Pixillation* (1970) featured a wide range of effects made by contrasting geometric forms with organic motion. John Whitney worked in collaboration with Jack Citron at IBM to make a number of films including the innovative *Permutations*. This film expresses Whitney's ideas about relationships to music and abstract form by permuting an array of dots into infinite kinetic patterns. Other artists working with computer animation around this time were Larry Cuba, Peter Foldes, and John Stehura.

The paths of contemporary animation and software development often overlap. The 3D visualization of the Death Star in *Star Wars* (1977) was one of the first uses of computer-generated animation in a feature film. Custom software was written to produce a wire-frame fly-through of the massive ship. Interest in computer animation briefly peaked with Disney's *Tron* in 1982, but soon receded due to the film's commercial failure. The industry gradually rebuilt itself into its current role as a major force in contemporary film. Pixar, the hugely successful animation studio that produced *Toy Story* and *The Incredibles*, operated for many years as a software development company. Pixar's RenderMan software (1989) enabled the rendering of 3D computer graphics as photorealistic images. RenderMan became an industry standard and Pixar continues to develop custom software for each film. The success of the company's films reflects its successful marriage of technical virtuosity and masterful storytelling.

Creating unique and experimental animation with software is no longer restricted