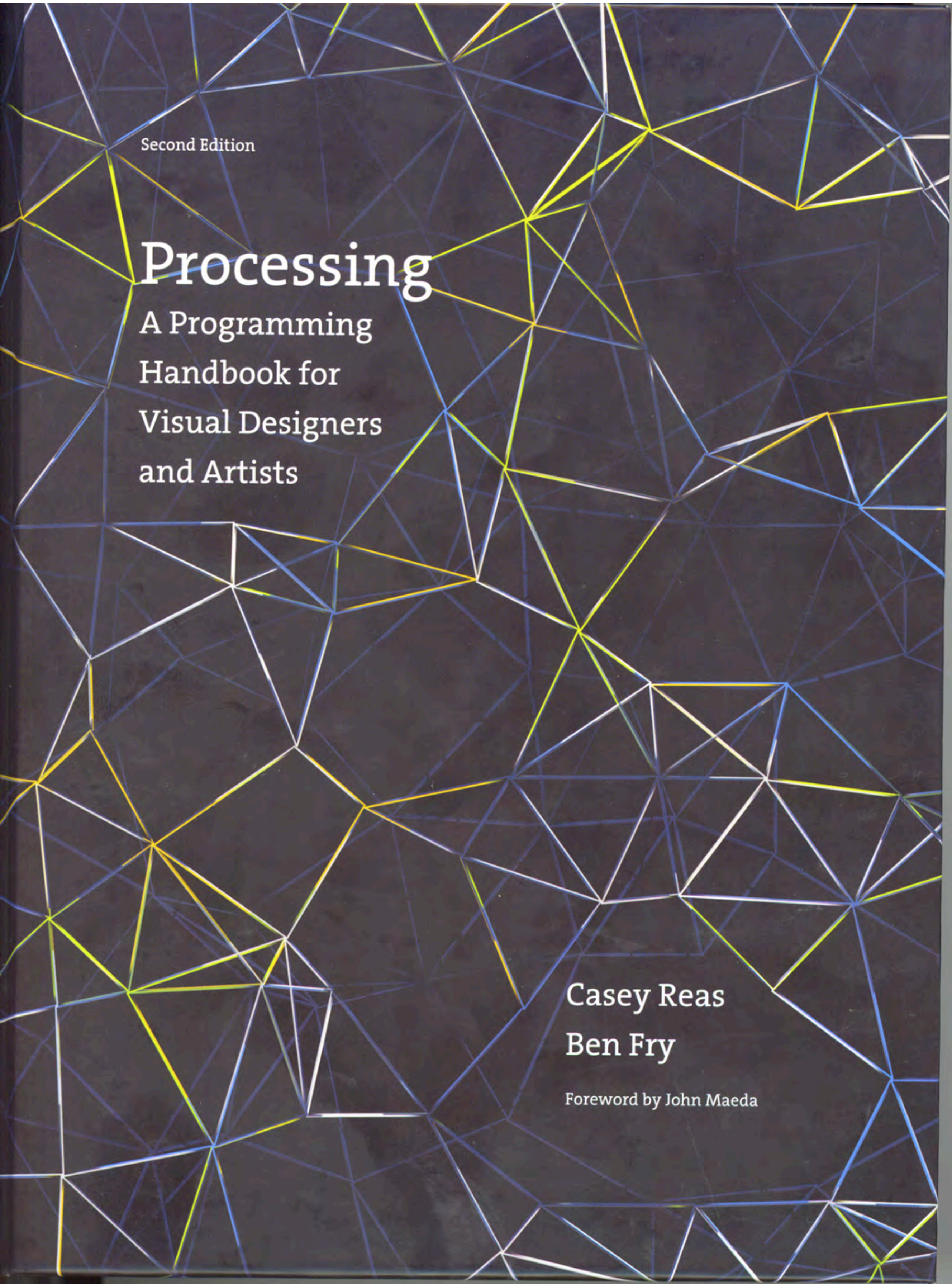


Second Edition

Processing
A Programming Handbook for Visual Designers and Artists

Casey Reas
Ben Fry

Foreword by John Maeda

This book was set in TheSerif and TheSansMono from LucasFonts by Toppan Best-set Premedia Limited. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication

Reas, Casey.
Processing : a programming handbook for visual designers and artists / Casey Reas and Ben Fry.—Second edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-262-02828-8 (hardcover : alk. paper) 1. Computer programming. 2. Computer graphics—Computer programs. 3. Digital art—Computer programs. 4. Art—Data processing. 5. Art and technology. I. Fry, Ben. II. Title.

QA76.6.R4138 2014
006.60285—dc23 2014016283

10 9 8 7 6 5 4 3 2

28 Arrays

This chapter introduces arrays of data.

Syntax introduced:

`Array, [] (array access), Array.length, printArray()`
`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 Boston 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, and so on. Arrays might store vertex data for complex shapes, recent keystrokes from the keyboard, or data read from a file.

For instance, an array can store five integers (1919, 1940, 1975, 1976, 1990), the years to date that the Cincinnati Reds won the World Series, rather than defining five separate variables. 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 an important detail for many programming languages. 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 to construct a bar chart.

x	50	61	83	69	71	50	29	31	17	39
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

The following examples to draw this chart demonstrates some of the benefits of using arrays, like avoiding the cumbersome chore of storing data points in individual variables. Because the chart has ten data points, inputting this data into a program requires either creating 10 variables or using one array. The code on the left demonstrates using separate variables. The code on the right shows how the data elements can be logically grouped together in an array.

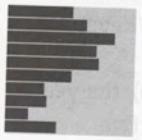
Separate variables

```
int x0 = 50;
int x1 = 61;
int x2 = 83;
int x3 = 69;
int x4 = 71;
int x5 = 50;
int x6 = 29;
int x7 = 31;
int x8 = 17;
int x9 = 39;
```

One array

```
int[] x = { 50, 61, 83, 69, 71,
            50, 29, 31, 17, 39 };
```

Using what we know about drawing without arrays, ten variables are needed to store the data; each variable is used to draw a single rectangle. This is tedious:



28-01

```
int x0 = 50;
int x1 = 61;
int x2 = 83;
int x3 = 69;
int x4 = 71;
int x5 = 50;
int x6 = 29;
int x7 = 31;
int x8 = 17;
int x9 = 39;

fill(0);
rect(0, 0, x0, 8);
rect(0, 10, x1, 8);
rect(0, 20, x2, 8);
rect(0, 30, x3, 8);
rect(0, 40, x4, 8);
rect(0, 50, x5, 8);
rect(0, 60, x6, 8);
rect(0, 70, x7, 8);
rect(0, 80, x8, 8);
rect(0, 90, x9, 8);
```

In contrast, the following example shows how to use an array within a program. The data for each bar is accessed in sequence with a `for` loop. The syntax and usage of arrays is discussed in more detail in the following pages.

Define ar

Arrays are c
brackets, [
(Each array
created wit
allocates sp
created, the
assign array
five elemen
different wa
setup().

```
int[] data

void setup
  size(10
  data =
  data[0]
  data[1]
  data[2]
  data[3]
  data[4]
}

int[] data

void setup
  size(100
  data[0]
  data[1]
  data[2]
  data[3]
  data[4]
}
```

```
int[] x = { 50, 61, 83, 69, 71, 50, 29, 31, 17, 39 };
fill(0);
// Read one array element each time through the for loop
for (int i = 0; i < x.length; i++) {
  rect(0, i*10, x[i], 8);
}
```

28-02

Define an array

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. (Each array can store only one type of data.) After the array is declared, it must be created with the keyword new, just like working with objects. 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 that explain 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 technique for creating and assigning elements of the array relates to setup().

```
int[] data;                   // Declare

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;
}
```

28-03

```
int[] data = new int[5];   // Declare, create

void setup() {
  size(100, 100);
  data[0] = 19;              // Assign
  data[1] = 40;
  data[2] = 75;
  data[3] = 76;
  data[4] = 90;
}
```

28-04

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

void setup() {
  size(100, 100);
}
```
28-05

Although each of the three previous examples defines an array in a different way, they are all equivalent. They show the flexibility allowed in defining the array data. Sometimes, all the data a program will use is known at the start and can be assigned immediately. At other times, the data is generated while the code runs. Each sketch can be approached differently using these techniques.

Arrays can also be used in programs that don't include a `setup()` and `draw()`, but the three steps to declare, create, and assign are needed. If arrays are not used with these functions, they can be created and assigned in the ways shown in the following examples.

```
int[] data;           // Declare
data = new int[5];    // Create
data[0] = 19;         // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```
28-06

```
int[] data = new int[5];    // Declare, create
data[0] = 19;               // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```
28-07

```
int[] data = { 19, 40, 75, 76, 90 };  // Declare, create, assign
```
28-08

Read array elements

After an array is defined and assigned values, its data can be accessed and used within the code. An array element is accessed with the name of the array variable, followed by brackets around the element position to read.

assign 28-05

t way, they
ta.
assigned
sketch can

draw(), but
ed with
following

28-06

28-07

assign 28-08

used within
, followed by

28-09
```
int[] data = { 19, 40, 75, 76, 90 };
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);
```

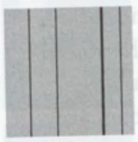
Remember, the first element in the array is in the 0 position. If you try to access a member of the array that lies outside the array boundaries, your program will terminate and give an `ArrayIndexOutOfBoundsException`.

28-10
```
int[] data = { 19, 40, 75, 76, 90 };
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 is accessed with the dot operator (p. 363–379). The following example demonstrates how to utilize it.

28-11
```
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
```

Usually, a `for` loop is used to access array elements, especially with large arrays. The following example draws the same lines as code 28-09 but uses a `for` loop to iterate through every value in the array.



28-12
```
int[] data = { 19, 40, 75, 76, 90 };
for (int i = 0; i < data.length; i++) {
  line(data[i], 0, data[i], 100);
}
```

A `for` loop 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()`.

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 Processing reference included with the software.

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

28-18

```
String[] trees = { "ash", "oak" };
append(trees, "maple"); // INCORRECT! Does not change the array
printArray(trees); // Prints [0] "ash", [1] "oak"
println();
trees = append(trees, "maple"); // Add "maple" to the end
printArray(trees); // Prints [0] "ash", [1] "oak", [2] "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");
// Prints [0] "ash", [1] "oak", [2] "maple", [3] "beech"
printArray(moretrees);
```

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

28-19

```
String[] trees = { "lychee", "coconut", "fig" };
trees = shorten(trees);  // Remove the last element from the array
printArray(trees);  // Prints [0] "lychee", [1] "coconut"
println();
trees = shorten(trees);  // Remove the last element from the array
printArray(trees);  // Prints [0] "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 at a time. 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.

28-20

```
int[] x = new int[100];  // Array to store x-coordinates
int count = 0;  // Positions stored in array

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

```
void draw() {                                                  28-20
  x[count] = mouseX;            // Assign new x-coordinate to the array    cont.
  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
  }
}
```

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` loop. 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 here.

28-21

```
String[] north = { "OH", "IN", "MI" };
String[] south = { "GA", "FL", "NC" };
arrayCopy(north, south); // Copy from north array to south array
printArray(south); // Prints [0] "OH", [1] "IN", [3] "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
printArray(west); // Prints [0] "MA", [1] "NY", [2] "RI"
```

New functions can be written to perform operations on arrays, but arrays behave differently than data types such as `int` and `char`. As with objects, 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 point to the same data. Changes made to `d[]` on line 14 modify the value of `data[]` in the `setup()` block. The `draw()` function is not used because the calculation is made only once and nothing is drawn to the display window.

```
float[]

void set
  halve(
  printl
  printl
  printl
  printl
  printl
}

void halv
  for (i
    d[i]
  }
}
```

Changing a
some addit
function as
copied in th
array is ret

```
float[] d
float[] h

void setu
  halfDat
  println
  println
  println
  println
  println
}

float[] ha
  float[]
  arrayCo
  for (in
    numbe
  }
  return
}
```