UNIT 16. ARRAY BASICS

Summary

You will learn a new fundamental programming technique called arrays. Specifically, you will learn

- How to use arrays to overcome the limitations of creating individual variables
- How to make and use arrays in Processing
- How to use arrays with loops to quickly solve large problems

Learning Objectives

After finishing this unit, you will be able to …

- Create an array with any data type and size
- Store data in an array, and retrieve data from it
- Iterate through arrays with a loop
- Get the length of any array
- Initialize an array with literals

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

 © James Young, 2016

16.1 **Introduction**

Arrays are a fundamental programming technique used to address the limitations of creating individual variables. A great way to really understand what this means is to work through a quick motivating example.

Let's make a quick processing program to create a particle (small point) that fires off from the mouse pointer at a random speed in a random direction. If it hits the edge of the screen, start over from the mouse in a new direction at a new speed.

Our global variables include finals for `MAX_SPEED` (I set it to 5). In addition, the particle needs global variables to keep track of its current position, `ballX`, `ballY`, and variables to keep track of its speed in the `x` and `y` directions, `ballSpeedX`, and `ballSpeedY`.

In the setup, when the program starts, set the ball position to be the center of the screen, and generate random speeds for the X and the Y. **note:** make sure to include negative X and Y to go left or up. In my case, I created a function to help out:

```
float randomInRange(float min, float max)
{
  float range = max-min;
  float r = random(range);
  return r + min;
}
void setup()
{
  size(500, 500);
  ballX = width/2;
  ballY = height/2;
  ballSpeedX = randomInRange(-MAX_SPEED, MAX_SPEED);
  ballSpeedY = randomInRange(-MAX_SPEED, MAX_SPEED);
}
```

From here, the draw block is straight forward. Clear the background, move the ball (add the `speedX` and `speedY` to the position variables), and draw the ball. If it's off the edge of the screen, then start again at the mouse and generate a new random speed and direction. Here is my whole program:

```
final float MAX_SPEED = 5;
float ballX;
float ballY;
float ballSpeedX;
float ballSpeedY;
```

 © James Young, 2016

```
float randomInRange(float min, float max)
{
  float range = max-min;
  float r = random(range);
  return r + min;
}

void setup()
{
  size(500, 500);
  ballX = width/2;
  ballY = height/2;
  ballSpeedX = randomInRange(-MAX_SPEED, MAX_SPEED);
  ballSpeedY = randomInRange(-MAX_SPEED, MAX_SPEED);
}

void draw()
{
  background(0);
  if (ballY < 0 || ballY > height ||
      ballX < 0 || ballX > width)
  {
    // move the ball to the mouse
    ballX = mouseX;
    ballY = mouseY;

    ballSpeedX = randomInRange(-MAX_SPEED, MAX_SPEED);
    ballSpeedY = randomInRange(-MAX_SPEED, MAX_SPEED);
  }

  ballX += ballSpeedX;
  ballY += ballSpeedY;

  stroke(255);
  point(ballX, ballY);
}
```

This is interesting, but wouldn't it look cool if we had more particles? Let's try to

update the program to three particles. To do this, we need to keep track of the positions and speeds of EACH particle. So, we need a bunch of new variables. I renamed my existing variables with a number for clarity:

```
float ballX1;
float ballY1;
float ballSpeedX1;
float ballSpeedY1;

float ballX2;
float ballY2;
float ballSpeedX2;
float ballSpeedY2;

float ballX3;
float ballY3;
float ballSpeedX3;
float ballSpeedY3;
```

Following, we need to repeat all the logic in the draw and setup blocks for the 2nd and third ball. We can cleverly use some functions to simplify this a little. However, there is only a limit to this, and it is very clunky. Try it on your own to really get a sense of this problem.

What if we wanted to have 100 particles? 1000? This quickly becomes crazy. Clearly there must be a better way – arrays! We will come back to this example later. For now, let's learn the fundamentals of arrays.

16.2 **What is an array**

An ***array is an ordered list of data of a given type.*** In English, that means a collection of a bunch of variables of the same type, where the collection is ordered. For example, you can have an array of 100 integers, an array of 10,000 strings, an array of 5 characters, and so on. In each case, you should imagine these collections as big lines of data, with one variable after another.

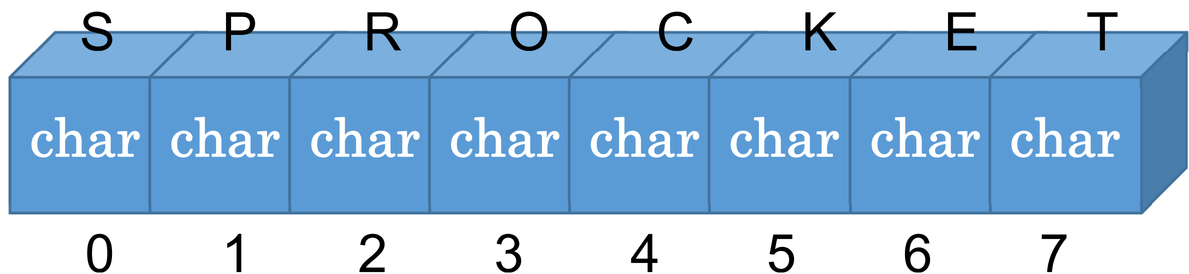
The power of arrays is that you can easily wrangle (create and use) large numbers of variables. Up until now, if you wanted 10,000 variables, you would need to type every one in and give every one a name. Using arrays, you can create 10,000 variables as easily as you can create 10!

We have actually seen an array before. When we learned strings, we learned that, under the hood, it just is a bunch of characters. At the core, a string is an array of characters. As shown in the diagram, the String `"SPROCKET"` is actually made up

 © James Young, 2016

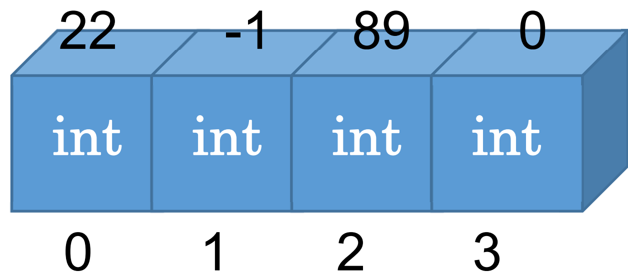
of 8 characters lined up in a row. This is an **ordered list of data** as we talked about above.

When we learned this in the strings module, we actually already learned a lot about arrays. They have a given length. ***Each bin in the array has a designated number called an index***, and the numbering starts at 0. In this case, it is an array of characters of size 8.



Unlike strings, with general arrays we are not limited to characters, or, working with arrays through an interface like the `String` type. We can make arrays of any type, and, can work with them directly.

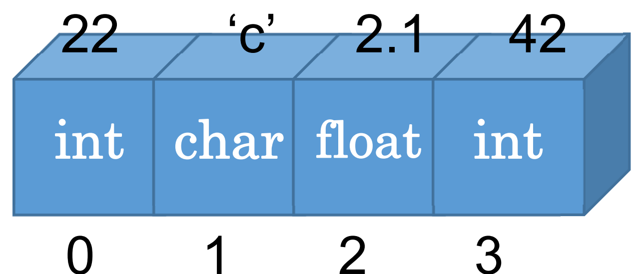
An array of four integers may look like the image here. As with strings, again, each box has its own unique number. The integers are all lined up. The first index is 0.



While arrays are great and quite flexible, there are some important key limitations:

Arrays are fixed length: Once an array is created, the length is fixed. When you create an array you decide how big it should be. This cannot change no matter what! BUT – what if you need an array to shrink or grow? This is a classic final exam question. There is only one way. Make a new array, and copy your data from the old one into the new one.

Arrays are homogenous: Every bin in an array is the same type. If you have an array of integers, every bin in the array stores an integer. You cannot have an array like in the inset. You have to choose the array type when you are creating it, and it cannot change.



Arrays are homogenous, ordered, lists of data of a single type. Let's learn how to make them.

 © James Young, 2016

16.3 **Creating Arrays**

Arrays work differently from anything else we learned in the course so far, and this comes up in many places. This is one of them. Unlike other variables, which only have one step, creating an array has two steps:

- ***Declaration:*** Declare the array variable container (as with other variables)
- ***Instantiation:*** Create the array (allocate memory) and assign it to the container.

Here is the syntax. To declare an array:

```
type[] variableName; // type is any Processing type.
```

You use the square brackets `[]` after a type to tell Processing that this is an array variable. For example:

Declare:

```
float[] studentScores;
String[] studentNames;
```

IMPORTANT. Array variables do not store the actual array (more on this below). It just remembers where in the computer memory the array is stored. As such, you do not specify the array size when you create the variable.

Once the array is declared, you need to instantiate it. This reserves the computer memory for all those bins you want.

Instantiate:

```
variableName = new type[arraySize];
```

`variableName` must already be declared, as in the previous step. You create a new array using the `new` keyword, and tell it the type and size. Once the new array is created, the variable stores where in computer memory the array is. At this point, this may seem like too much information, but this computer-memory approach helps to explain and understand a lot of array quirks.

Here is another example:

```
double[] studentScores;
studentScores = new double[30]; // 30 students in class
```

Of course, just like with other variables, you can combine the **declaration** and

 © James Young, 2016

instantiation into one command:

```
int[] studentAges = new int[30]; // declare and instantiate
```

Remember, once the array is instantiated, the size cannot be changed!!

Here are some more examples of valid array declarations and instantiations. Any valid Processing type can be used. Be sure that the type of the variable (**declaration**, on the left) matches the type of the array (**instantiation**, on the right), or it will not work.

```
double[] gameScores   = new double[5000];
String[] blogComments = new String[200];
long[] bigNumbers     = new long[100];
```

remember: you must **declare** and **instantiate** or you cannot use the array.

Although we haven't yet seen how to use arrays, now is a good time to highlight the power of the above examples. In each case, we are creating a large number of variables (5000, 200, and 100) without a lot of typing.

Finally, what is the maximum size of an array? This depends on the language. In Java, it's the maximum integer value, around 2 billion. In practical use, however, you won't use arrays this large. For bigger data sets you will use other techniques.

16.4 **Using Arrays**

So now we can create an array of any type and any size, but how can we actually use it in our code?

You can access any bin in the array just by using the array variable name, and putting the bin number in brackets:

```
arrayVariable[binNumber]
```

In this case, the result is that you have a variable that matches the array type.

For example:

```
int[] numbers = new int[10]; // array of 10 ints
numbers[0]  ← first int in the array
numbers[5]  ← sixth int in the array
numbers[9]  ← last int in the array
numbers[10]  ← ?? error
```

      © James Young, 2016

That last line will raise an error: Index out of bounds. The bin doesn't exist. This is the famous off-by-one error we have been seeing all course, so be careful!

Using the above syntax, you can now use the variable just like any other. For example, to store values in the array, you use the array variable and bin combination just like any other variable:

```
variableName[bin] = value
numbers[3] = 10;
numbers[0] = -1000;
```

Since this is an array of integers, you can store any valid `int` in the array bins.

Likewise, we can retrieve the values from the array bins just like any other variable:

```
int i = numbers[4];
if (numbers[1] < 100) { …
```

You can use this array variable – with the bin number in brackets – anywhere that a normal variable can be used.

When accessing the array bins, you can give any integer as the bin number. This could be a variable, a literal, even a calculation as long as it results in an integer. For example, this is valid:

```
int i = 5;
values[i] to access bin i
```

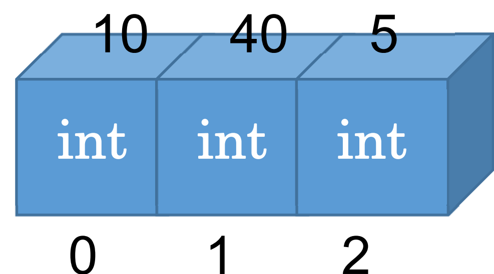
but it does not make sense to use non-integers:

```
values[5.5] ← doesn't make sense
values["yo!"] ← doesn't make sense
values[false] ← doesn't make sense
```

So now you have the tools to make large collections of variables, and to use them. It may not be obvious how all this fits together, so let's work through some examples.

16.5 **Example: first arrays**

Make an array of integers, size three, and put three numbers in it. Put the numbers 10, 40, and 5, so that the array looks like this:



Now, make an array of three strings and put three

 © James Young, 2016

messages into it: `"woah!"`, `"hey there"`, `"nice day"`.

Here is my code for this so far. First I create the variables and declare the arrays. Then, I store the data into the arrays.

```
final int COUNT = 3;
int[] sizes = new int[COUNT];
String[] messages = new String[COUNT];

sizes[0] = 10;
sizes[1] = 40;
sizes[2] = 5;

messages[0] = "woah!";
messages[1] = "hey there";
messages[2] = "nice day";
```

Use the integer array to set font sizes for the three messages, and draw it to screen.

Since the text is all different sizes, it's a little tricky to draw them to not overlap. My technique is to remember the line position (`y`), and before drawing the text, add the text size. This way, you move DOWN by the text size, and it draws UP from the position, into the space you just filled:

```
int y = height/2;
y += sizes[0];
textSize(sizes[0]);
text(messages[0], 0, y);

y += sizes[1];
textSize(sizes[1]);
text(messages[1], 0, y);

y += sizes[2];
textSize(sizes[2]);
text(messages[2], 0, y);
```

By using arrays, we were able to greatly simplify the variable creation process. Instead of copy-pasting to create three variables, we just created them once. However, we still copy-pasted the code to use the variables. If this example had 100 items, it would be a lot bulkier. Let's learn how to use `for` loops to simplify this.

 © James Young, 2016

16.6 **Iterating Through Arrays**

We can solve the above problem quite simply. Since the array index – the bin number – is an integer, we can use a variable there. Since we can use a variable, we can use a `for` loop to iterate through that variable! Piecing it all together, we can use a for loop to go through the bins to simplify the above code.

Notice how the code to actually draw the messages on the screen was identical, with the only change being the bin number used in the array? So, the above three drawing commands are equivalent to the following:

```
int y = height/2;
for (int i = 0; i < COUNT; i++)
{
  y += sizes[i];
  textSize(sizes[i]);
  text(messages[i], 0, y);
}
```

The `for` loop goes through 0, 1, 2, giving us the same output result as earlier.

The real power of arrays is in how nicely they scale up. In this case, let's add two more lines. To do this, all that we need to do is

- Make the arrays bigger
- Store more data in the arrays

Try it out yourself. The `for` loop does not need to change, only the arrays.

16.7 **Example: random points**

Let's work through an example to illustrate the power of arrays. This program will generate 8 random points around the screen on startup. Then it will draw lines from those points to the mouse each frame. This will look like the points are stuck but the mouse can move. Finally, if the mouse button is pressed, generate a new set of random points.

First, let's create our arrays as globals. We should use a `final int` to determine the size of the array, and make an array for the `x` coordinates, and another array for the `y` coordinates. These need to be global so that they persist and do not get erased each time we draw:

 © James Young, 2016

```
final int POINTS = 8;
final int[] x = new int[POINTS];
final int[] y = new int[POINTS];
```

Now, we will need to generate random points on startup, AND, if the mouse is clicked. Instead of writing this code in two places, let's make a function to do it for us. This function should use a `for` loop to go through all the bins of the `x` and `y` arrays. Then, at each bin, store a random number. For `x`, we want this number to be within the range `0<=x<width`, and for `y`, we want `0<=y=height`.

```
void newPoints()
{
  for (int i = 0; i < POINTS; i++)
  {
    x[i] = (int)random(width);
    y[i] = (int)random(height);
  }
}
```

This code goes through all the bin numbers from `0` to `POINTS-1`, and creates a random `x` and `y` for those bins. Each bin gets a new random number.

Call this function in the startup code for the initial setup. And, in `draw`, use an `if` statement to check if the mouse is pressed, and if so, call this again.

Similarly, instead of putting all your drawing code into the draw block, make a new function called `drawLines`.

```
void drawLines()
{
  for (int i = 0; i < POINTS; i++)
   {
     line(mouseX, mouseY, x[i], y[i]);
   }
}
```

The code here goes through all bin numbers `0<=i<POINTS`, then draws a line from the mouse to that line.

Finally, the draw is simple. We clear the background, set the stroke, check the mouse pressed (and get new points if needed), and draw the lines:

 © James Young, 2016

```
void draw()
{
  background(0);
  stroke(255);
  if (mousePressed)
    newPoints();
  drawLines();
}
```

Run the program and you can see that we have 8 random points that the mouse draws lines to. Without arrays, imagine all the work this would have taken! We would have needed a lot more variables, and we wouldn't have been able to use the `for` loops for creating the random points and drawing the lines.

Now, scale the example up to 80 points. All that you need to do for this to happen, is to change the `POINTS` global, and everything else works!! Easy! 8 points is the same programming work as 80! Or 800!

16.8 **Example: mouse explosion**

Remember the example at the beginning of the unit? The one with a point shooting out from the mouse at a random speed and direction? Go back and type that up again – we're going to make it awesome.

So, let's see if we can modify the example to use arrays. First, let's make a global variable to tell us how many balls we need. For now, just set it to `10`. Next, *upgrade* those floats for the ball position and speeds to arrays, since we will need a unique position and speed per ball!

This is what I have:

```
final int BALLS = 10;
float[] ballX = new float[BALLS];
float[] ballY = new float[BALLS];
float[] ballSpeedX = new float[BALLS];
float[] ballSpeedY = new float[BALLS];
```

This now gives us 10 sets of variables, enough for 10 balls. Previously we only had one.

Now the program won't work, because we were using our variables, `ballX`, `ballY`, `ballSpeedX`, `ballSpeedY`, as floats, and now they are arrays.

The solution to this is that, each time we would do something to one of these

variables, we need to upgrade to using arrays. Previously we would do something to one variable (e.g., add movement speed to position), but now we need to do the same work for *every* bin in the array. This means that we need to use `for` loops.

At the setup, we placed the original ball at the screen center. So, let's place all the balls there. Also, we set a random speed to the ball, so we need a new random speed for each ball as well. Wrap the existing startup code in a `for` loop, and update the variables to use the array, such that you do the same operation for all bins in the array:

```
for (int i = 0; i < BALLS; i++)
{
  ballX[i] = width/2;
  ballY[i] = height/2;
  ballSpeedX[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
  ballSpeedY[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
}
```

Similarly, in the draw block we need to repeat all the logic for *each* ball. We need to check each ball if it's outside the screen, and if so, move it to the mouse. We need to move each ball by adding the speed variables to its location. We need to draw each ball.

Again, wrap the existing code in a `for` loop to save a lot of work instead of treating each ball individually. Make sure to update each call to the old variable with the new array and bin number.

```
for (int i = 0; i < BALLS; i++)
  {
    if (ballY[i] < 0 || ballY[i] > height ||
      ballX[i] < 0 || ballX[i] > width)
    {
      // move the ball to the mouse
      ballX[i] = mouseX;
      ballY[i] = mouseY;

      ballSpeedX[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
      ballSpeedY[i] = randomInRange(-MAX_SPEED, MAX_SPEED);
    }

    ballX[i] += ballSpeedX[i];
    ballY[i] += ballSpeedY[i];
```

 © James Young, 2016

```
    stroke(255);
    point(ballX[i], ballY[i]);
  }
```

Cool! It should work now. Try scaling up to 100, or even 1000 balls! To do this you only need to change one number.

16.9 **Array length**

When you create an array, you determine its length. That length is fixed and can never change. Also, knowing that length is useful, for example, when we use a `for` loop to go through an array.

So far, we use a named constant (a `final`) to set the size of the array, and use that same constant to determine how many bins are in the array. Sometimes, it's not so simple to keep track of how big an array is, or which global refers to which array. This is dangerous, because if you go off the end of an array – try to access a bin but the array is too small for that bin – your program crashes.

Processing provides a simple mechanism to get the length of an array. If we have an array variable, we can get the length with:

```
variableName.length
```

For example:

```
int numbers[] = new int[100];
println(numbers.length);
```

The output is 100, which is the number of bins in the array. As such, the last bin in an array is always `.length-1`. (remember! 0 based counting!).

This looks very similar to how we get the length of strings, but there is a very important difference.

For Strings:

```
String s = "Eric the Fruit Bat";
println(s.length());
```

And, for arrays

```
int[] numbers = new int[100];
println(numbers.length);
```

They look similar but there is a huge difference. Can you spot it?

When you ask a string for its length, you need to put brackets at the end of the `length()`. For arrays, you don't use the brackets: `.length`. This is really annoying, and stems from the fact that strings are Objects and use object-oriented programming. Arrays are built directly into the language and are not objects. Unfortunately, we don't learn much more about this until a later course.

This difference is very confusing and probably makes little sense at this point. Unfortunately, you ***just need to memorize this. Arrays, .length. Strings, .length().***

Even when you know the named constant for the length of an array, you are safer using the array length. This is bullet proof, and even if something shifted or there is a mistake, `.length` always gives you the correct array size.

16.10 **Example: make it rain!!!**

Let's make it rain!

Make an array where each bin represents a rain drop. The bin number can be the `x` coordinate of the rain drop (so they are in every column), and we can store a random `y` value in the bin so that they are at random heights.

If we have more rain drops than `x` coordinates, use modulo to make it wrap around. For example, given a canvas size of `500`, bin `500` is a rain drop also in the left-most column, since `500%500` is `0`.

A quirk of this example is that we will generate a random number of dots in the startup. This means that we can't rely on a hard-coded final constant for the array size. It also means that we have to separate the array initialization and declaration.

We need to use the array throughout the program, and it cannot lose its data every time we draw, so the variable needs to be global. However, we don't initialize it to create a new array until the startup. Let's make the global variables. In this example we also use lines for rain (for artistic effect) so let's define the line length as well

```
final int DROP_Y_LEN = 10;
int drops[];
```

In the startup, let's instantiate our array and actually make some dots.

 © James Young, 2016

To generate a random number, let's be sure that we have at least one per column (width). Then, add a random amount, up to some maximum. I used a global for this and set the global to 10,000. Use this calculated number to create your array. ***NOTE: Make sure that you make it an*** `int`***, since you need to use an*** `int` ***to set the array size.***

```
int count = (int)random(MAX_ADDITIONAL_DROPS)+500;
drops = new int[count];
```

Now, use a for loop to set each drop's `y` value to a random position. This is a great place to try out the new `.length`. Be careful, you want a new random position for each dot. If you do this wrong, each dot will have the same `y` coordinate.

```
for (int i = 0; i < drops.length; i++)
{
  drops[i] = (int)random(height);
}
```

OK, we are all initialized.

The `draw` loop is actually pretty simple. We clear the background, and then use a `for` loop to go through each bin in the array. Since the size of the array was only stored in a local variable in startup, we no longer have access to that! We must rely on the `.length` property.

Calculate the `x` coordinate of the drop (`x` mod width – if it's bigger than width, just wrap around). The `y` coordinate is just the data in the bin. Use the drop line height to draw your rain drops:

```
for (int i = 0; i<drops.length; i++)
  {
    int x = i;
    int y = drops[i];
    stroke(random(256));
    line(x,y,x,y+DROP_Y_LEN);
}
```

All that is left is to animate them. A simple way is to simply add some value to the `y` coordinate (i.e., actually modify the bins of the array) each time. Use modulo to wrap around if it goes off the edge of the screen. Get this working before moving forward.

I think it's more fun, though, if the drops fall at different speeds. A simple trick we can do is to make there appear to be layers. What if the 0th, 3rd, 6th, 9th, etc., and so on

 © James Young, 2016

moved at one speed, the 1st, 4th, 7th, 10th, etc., at another, and so on? We can do this with the following formula. Try to figure it out as an exercise

```
drops[i]=(drops[i]+i%3+1)%height;
```

Now that you're done, try to spice it up. What about color? Before each line, try this

```
stroke(random(256),0,random(256));
```

16.11 **Array Initialization with Literals**

Often times, as we have done, you want to pre-load an array with values. For example, we could use an array of strings to store the days of the week for the header of a calendar.

```
String[] days = new String[7];
days[0] = "S";
days[1] = "M";
days[2] = "T";
days[3] = "W";
days[4] = "R";
days[5] = "F";
days[6] = "S";
```

This is very useful, as now we can use this to draw the header of a calendar like we did in an earlier unit.

Go back and take a look (Section 13.2) at the code, type it up, we'll work on it. Of particular interest is the following ugly code:

```
// draw title bar
  int bottom = CAL_TOP+CAL_SPACE;
  int left = CAL_LEFT;
  text("S", left, bottom);
  left += CAL_SPACE;
  text("M", left, bottom);
  left += CAL_SPACE;
  text("T", left, bottom);
  left += CAL_SPACE;
  text("W", left, bottom);
  left += CAL_SPACE;
  text("R", left, bottom);
```

 © James Young, 2016

```
  left += CAL_SPACE;
  text("F", left, bottom);
  left += CAL_SPACE;
  text("S", left, bottom);
  left += CAL_SPACE;
```

We use a variable, left, to remember the leftmost spot for the letter. We draw a letter, then move the variable along.

Now that we have an array of days, we can replace all the above code with a simple for loop to get the equivalent output:

```
int bottom = CAL_TOP+CAL_SPACE;
int left = CAL_LEFT;
for (int i = 0; i < days.length; i++)
{
  text(days[i], left, bottom);
  left += CAL_SPACE;
}
```

Which is much nicer! However, we still have that bulky mess when we created the array – we had to set every bin individually.

When we create an array and want to store data into it right away, we can do what is called a literal initialization of an array. This is a shortcut that

- Creates a new array in memory. We don't need the new command.
- Populates the array with the data we want. We don't need to set every bin individually.

There is special syntax for this, and it looks like the following:

```
type[] variable = {element, element, element, …};
```

You just use the squiggly brackets `{` and `}`, and put your data in a list, separated by commas. In our above example, we can simply make our days array as follows:

```
String[] days = {"S", "M", "T", "W", "R", "F", "S"};
```

Processing automatically detects the length, allocates the memory for you, puts the data in the memory, and stores that memory address in days. All in one statement.

For example, what if I were to ask you to draw lines between the following sets of `x,y` points?

 © James Young, 2016

```
(100, 40) -> (140, 160) -> (40, 80) -> (160, 80) -> (60, 160)
-> first point
```

Previously you had to hard code this in a bunch of line statements. With arrays, maybe you could put them into arrays, and use a `for` loop to draw it. Previously, setting up the arrays would hardly be worth the effort. But with literal initialization, this gets much easier.

Let's make two arrays and initialize them with literals with the above numbers:

```
int[] xPoints = {100, 140, 40, 160, 60};
int[] yPoints = {40, 160, 80, 80, 160};
```

Now, we can draw them simply with a `for` loop. Let's go through all the bins, and draw a point from the current bin to the next one. We can use modulo to wrap around, so that when we go off the right edge of the array it goes back to zero:

```
for (int i = 0; i < xPoints.length; i++)
  {
    int next = (i+1)%xPoints.length;
    line(xPoints[i], yPoints[i], xPoints[next],
    yPoints[next]);
  }
```

What was the output?

Check your Understanding

16.12 Check Your Understanding: Exercises

Exercise 1. Create an array of floats with ten thousand bins, and practice both putting and retrieving data into and from the array.

a. Create the array with 10,000 bins.
b. Store the number 9999 in bins 0, 1, and the last bin.
c. Use `println` to print out the data in bins 0, 1, and the last bin

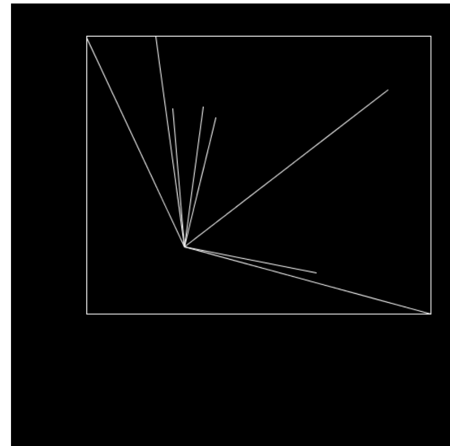
Exercise 2. Generate 500 random points (use two array variables, one for `x` and one for `y`), and draw them. Generate the points once, in the setup, and draw them each block.

a. Make the points move slowly to the right
b. Draw a line through the points, that is, from point 0 to 1, 1 to 2, etc.

 © James Young, 2016

Exercise 3. Make a program to plot the following points on a 500x500 canvas. Note: use literal array initialization and `for` loops.

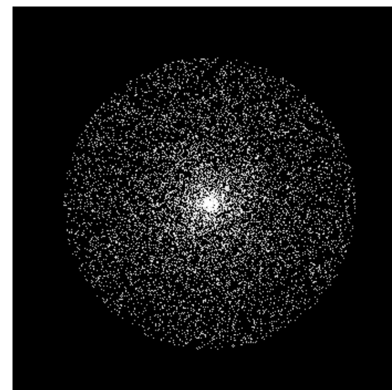
```
(150, 120), (158, 149), (166, 174), (175, 195), (183, 211),
(191, 224), (200, 233), (208, 240), (216, 245), (225, 247),
(233,  249), (241, 249), (250, 250), (258, 250), (266, 250),
(275, 252), (283, 254), (291, 259), (300, 266), (308, 275),
(316, 288), (325, 304), (333, 325), (341, 350), (350, 380)
```



Exercise 4. Update example 16.7 to draw a best-fit box around all the points, as shown here. This takes a little bit of work. The way to do this, is to find the smallest `x` and the smallest `y` in both the x and the y arrays, and then the largest `x` and `y`. Then, draw a box from the `smallestX,smallestY` to the `largestX,largestY`, and you have the box. How can you find the smallest and largest elements in an array? You need to do it manually, using `for` loops.

Exercise 5. Update example 16.8, the one with the points exploding from the mouse. Currently, the particles fire off randomly in any direction. Add gravity so that the balls fall toward the bottom of the screen, so that it looks like a fountain. Instead of thinking about gravity moving stuff (gravity doesn't move things, it accelerates them!), think about gravity accelerating things toward the ground. Therefore, each frame, you should add an amount of gravity to the ball's Y speed, not the position. Since gravity is $9.2m/s^2$, and we have `60` frames a second, you should add $9.2m/s^2 * 1s/60f = 0.153m/f$. Let's assume 1 pixel is 1 meter ☺, so you need to add 0.153 each frame to the speed.

1 metre per pixel! Wow I wish my monitor was that big.



In this exercise, you will animate the Big Bang at the start of the Universe! Draw a lot of "stars" (just single white pixels against a black background). Try 2500 stars to start with (but make that a named constant, of course). All the stars will start at exactly the same point (the center of the canvas – clearly the center of the universe), then they will "explode" outward, each with its

 © James Young, 2016

own random speed and direction.

For each star, you will need to know:

- Its current location (both X and Y)
- Its speed and direction of motion, which should be stored as separate X and Y motions (the change in X and change in Y to use each frame).

Since you have many stars, you will need to use arrays, and since you need to remember the location between frames, use global variables.

In the setup function, initialize all of the elements in all four arrays. The positions are easy: every star should start at the center of the canvas. For the velocity of each star, generate a random angle `theta` (between `0` and `2*PI`), and a random speed (from `0` to some maximum speed – try `3.0`, and of course use a named constant for this). Split the speed into separate `x` and `y` velocities by multiplying the speed by `sin(theta)` and `cos(theta)`, in the usual way. Note: It's a mistake to try to avoid `sin`/`cos` by generating separate random numbers for the `x` and `y` velocities. Do you see why? Try it! You get a rather strange universe that way.

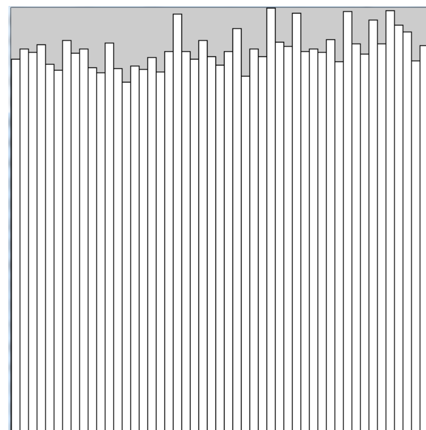
In `draw`, draw each star as a point, and then move it to the next position by adding its velocities to its position. For an even better effect, give each star a different color. Add some randomness each time you draw a star so that they'll "twinkle".

Boom!

a. For the ultimate effect, use the perspective technique from the exercise in unit 14 to make the explosion 3D! Generate `(x,y,z)` locations, use velocities in a random 3D direction, and apply perspective.

Exercise 6. Update example 16.10 to make the rain go in an angle. Make it both draw the line in an angle, and, move in an angle. This is a little tricky, but the array component should be easy.

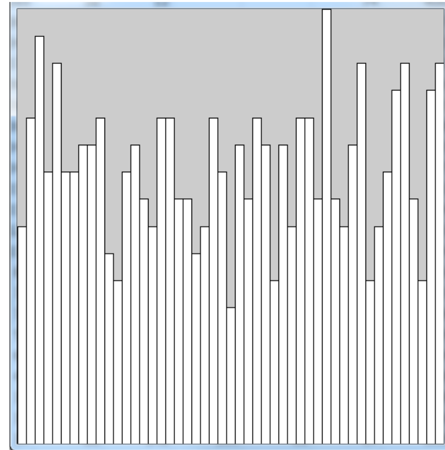
Exercise 7. In the real-life Lotto 6/49, as of Nov. 16, 2015, a total of 19,920 numbers between 1 and 49 have been generated, in 3,320 draws held since 1982(**) (not including the "bonus" numbers). The top histogram(*) (bar graph) on the right shows how many times each of the 49 numbers have been drawn. The number 28 was drawn only 378 times, but the number 31 was drawn 450 times. Does this mean that the game isn't random? That 28 is unlucky? That 31 is lucky? Let's run a simulation and compare it to



 © James Young, 2016

real life.

The program will generate `NUM_DATA_ITEMS` (say, 500) random numbers between `1` and `MAX_NUMBER` (49). Then it will count how many times each possible number was generated (the number's frequency). Finally, it will display a very simple histogram showing the results, as shown at right. The bottom histogram shows one possible result when only 500 numbers were generated. It looks a lot more "random" doesn't it? You'll get a different result every time.



Complete a `void generateData()` function. It should create an array of `NUM_DATA_ITEMS` integers, each from `1` to `MAX_NUMBER` (note: 1 not 0!) and store it in the global variable `theData`.

Complete a `void findFrequency()` function. It should create an array of `MAX_NUMBER` integers and store it in the global variable `frequency`. It should count the number of times that each integer `x` occurs in `theData`, and store that count in `frequency[x-1]`. (The -1 is because the lowest number is 1, not 0.)

Finally, complete a function `plotHistogram()` which will draw a histogram of the data in the frequency array. It will first need to scan through that array and find the maximum value that appears. The bar for that element should be the full height of the canvas, and all other numbers should be scaled accordingly. The full width of the canvas should be divided into `MAX_NUMBER` bars. The bottom of all of them should be at the bottom of the canvas.

Just for fun: Change `NUM_DATA_ITEMS` to `19920` and try it again. How do your results compare to the real Lotto 6/49 results? Are the real Lotto 6/49 numbers similar to your "random" ones? Do you think the game is really random?

(*) For a discussion of Histograms, see the Wikipedia page at en.wikipedia.org/wiki/Histogram

(**) The Lotto 6/49 data comes from www.lotto649stats.com/position_frequency.html

Exercise 8. This question will use the perspective technique introduced in Unit 14 exercises, using perspective to simulate a 3D image. This time, you'll draw a field of "stars" (single pixels), and make them fly toward you, or away from you, by changing their `z` coordinate. (Remember, the `z` coordinate controls

© James Young, 2016

how far "away" or "into the screen" an object is.) The mouse will be the control for your spaceship – when mouseX is in the center of the canvas, the spaceship will be stopped, and the stars won't move. Moving the mouse to the right will make the stars move toward you (as if you were flying forward through space), and moving the mouse to the left will make the stars move away from you.

Use three float[] array variables, starX, starY, and starZ to hold the positions in 3D space of STAR_COUNT stars. The star with index i will have a virtual 3D position in space of (starX[i], starY[i], starZ[i]).

Write a function void generateStar(int i) which will create a random star and store its 3D location in the global variables starX[i], starY[i], starZ[i]. The stars should start with x and y coordinates from -width/2 to +width/2 and -height/2 to +height/2, respectively. (These are virtual coordinates, not canvas coordinates, and the virtual (x,y) point (0,0) – the "vanishing point" – will be in the center of the canvas. That's where the stars should come from, or go to.) The z coordinate should be between 0 and MAX_Z – a predefined constant giving the maximum possible z coordinate for visible stars.

Complete the setup function by creating the three arrays needed, and filling them with random star positions, using the generateStar function.

Create a void drawProjectedPixel(float x, float y, float z) function (modify from the Unit 14 example) to project and draw the point. As we want to make stars with large z coordinates look far away, use the z coordinate to control the color ("brightness") of the star. Stars at z=0 should be white, and stars at MAX_Z should be black, with all others at an intermediate shade of gray.

Complete draw to animate moving through a field of stars. Each frame, use drawProjectedPixel to draw all of the stars against a black background. Move every star by changing its z coordinate (x and y never change). All stars should change z by the same value from -MAX_SPEED (when the mouse is on the far right) to +MAX_SPEED (when the mouse is on the far left). Use an int not a float for the speed, so that it will be easier to make it 0 and stop the spaceship. Finally, regenerate the stars so that you never run out of them. When any star gets too close to draw (z≤0) or too far away to see (z>MAX_Z), use generateStar to create a new one to take its place.

a. Yes, a double gold exercise! Modify the star field so that you can rotate it in an arbitrary fashion, using the technique shown in Unit 14 exercises. The challenge here is how and when to apply the rotation. Hint: the star coordinates themselves should NEVER change, only when you project do you rotate.

 © James Young, 2016

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

 © James Young, 2016