

UNIT 17. ARRAYS AND MEMORY

Summary

You will learn a about how arrays are stored in computer memory, specifically:

- ♦ The difference between data in a variable and data in general memory
- ♦ Basics of how a memory address is stored, and how to say “no address”
- ♦ How this impacts basic array operations like copying and comparing



Learning Objectives

After finishing this unit, you will be able to ...

- ♦ Determine the default value stored in your arrays immediately after creation
- ♦ Use and compare variables against `null`
- ♦ Copy an array
- ♦ Compare an array

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.

17.1 Introduction

Unlike other variables we have used so far, to use an array, you cannot simply make a variable and then use it. We have a new step, we need to instantiate the array with the `new` keyword.

Why this new step? Why don't we do this with our other variables? Well, we don't do it with any of the primitives at all, because what is stored inside the variable is actually the data. This may sound obvious, but in the case of arrays, the **array data isn't actually stored in the variable, it's stored somewhere else**.

This is also the case for strings, and any objects once you learn object oriented programming. In Processing, a lot of the mess of Strings is hidden from you with nice syntax, but weird things are going on behind the scenes.

When you create an array variable, just like any variable, you need computer memory to store the data. For arrays, the computer doesn't know how much memory you need until you instantiate the array, because it doesn't know how many bins you will need. Therefore, it doesn't know how large to make the variable. The solution is the following:

- ♦ We store arrays in general computer memory (not in the variable), so we can decide how large it needs to be later.
- ♦ **The array variable only stores the address of where the array is in general memory.** We need to store this address to keep track of where the array is, after we create it
- ♦ We know how large an address is, so Processing is happy to create your variable.

To reiterate:

The array variable stores an address to the array. Until we give it an address, the variable is uninitialized, just like any other variable.

```
int[] intArray; // variable stores an address to an array.
```

When we know how large we want our array, we need to ask the computer for some memory.

The `new int[100]` command actually is asking the computer to go off and find us some empty memory, large enough to store 100 integers (how many bytes is this?). Once the computer finds this memory, reserves it (puts up a fence, marks it as taken), and gives us the address. Imagine the computer is going land shopping for you, finds just the right size, puts up tape so no one else can take it, and tells you where it is.

Processing stores that address in our variable.

```
intArray = new int[100];
```

To reiterate, the `new` keyword goes off and finds a chunk of memory big enough to store 100 integers in this case, and gives the address, which is then stored in `intArray`. Most of this time, this memory issue is invisible to you. When you use arrays, you don't usually think about memory addresses.

```
intArray[5] = 4;
```

However, the above command actually takes a trip out to the memory that you have saved, takes your 4, and stores it in the 5th bin in your memory.

We can actually take a peek at the memory address, just to prove a point. This is not only advanced, but not actually very smart – only really advanced programmers, and even then, only in very specific situations, look at memory addresses. For most of us, it's a waste of time. However, for the purpose of peeking at what is happening, here is a silly trick to get Processing to give us the memory address. Concatenate the array with a string.

```
println(""+intArray);
```

The memory address will be something weird, like `[I@1f3f158`. Computers have a very systematic way of storing memory, which doesn't really make sense to us.

As an example, when we create our variable, we just get a variable in our program.

```
int[] intArray;
```

However, when we create a new array:

```
intArray = new int[100];
```

The computer searches its memory, finds enough space for our 100 integers, and marks off the space. Every time we use `intArray`, it actually goes off to that address and works there. E.g.,

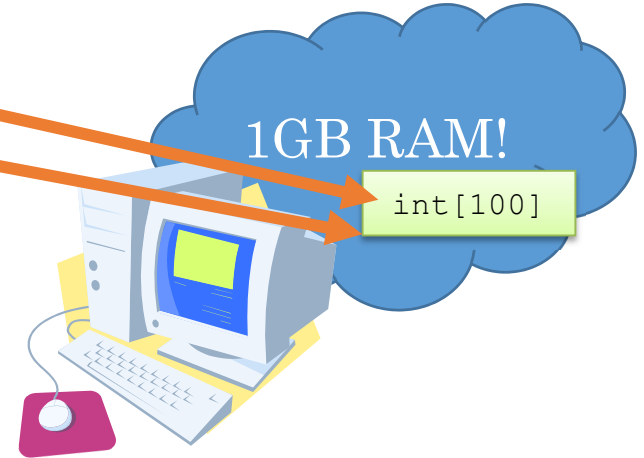
```
intArray[4] = 20;
```

Goes off to that plot of memory, puts a number 20 in bin 4, and comes back.

Here is a series of visuals of what is going on. Here is our modern computer with loads of ram.

```
int[] intArray;  
intArray = new int[100];  
intArray[4] = 20;
```

When the `intArray` variable is created, it has no memory address, and cannot be used. After `new` is called, then `intArray` stores the address of the section of memory we have to store our array data. Then, when `intArray` is used, the computer actually goes off and changes that memory.



Although I said we generally don't think about arrays and memory, there are some key times when we do, so you need to clearly understand this. Particularly when you start tossing arrays to functions, and wanting to do things like shrink or grow an array, you need these fundamentals down pat.

17.2 Default value of array entries

In Processing, what happens when we create a variable, do not initialize it, and then try to use it? For example:

```
int variable;  
println(variable);
```

If you try to run this, you get an error. Processing says “the local variable `variable` may not have been initialized.” This makes sense. Since we put nothing in the variable, what should the `println` do?

This is a feature of many languages. The compiler looks and can detect this instance, and refuse to use a variable that has never been set to a value.

Unfortunately, this isn't a trivial thing for the computer check. When working with arrays, it gets even harder. In fact, Processing just gives up. Let's try the following similar example. We create an array variable, instantiate the array, but do not put any data in the array bins:

```
int[] intArray = new int[10];  
println(int[0]);
```

This time it works! Huh? Why? That variable is not initialized!

Because it's hard for the computer to detect which array bins are initialized and which


are not, Processing has a workaround. When you create an array, it does a bit of work and fills the array with default values. This way, we know what is in the variable. If you imagine the land analogy I used earlier, when the `new` keyword finds and cordons off your new memory, it sends in a bulldozer to clear the memory to some default state.

What are these defaults? The defaults depend on the data type.

- ♦ For integer types, including the character, the default value is `0`. Careful (Advanced): for characters, this is the ASCII code `0` and not the character `'0'`.
- ♦ For Booleans, the default is `false`
- ♦ For Strings (and all objects), the default is `null`

Wait – what? `null`? What the heck is `null`? We'll get to that very shortly.


Generally be careful of relying on default values. That is, you may think: I know that my numbers get set to `0`, so I will rely on that fact. This is not a bad idea, but the problem is, other languages may treat this differently, (e.g., C or C++) creating a very hard to find bug! Be clearly aware of the parameters of your computer programming language.



Advanced: Depending on the compiler and run-time environment, C or C++ programs may not clear the array memory at all, and give you whatever data was there previously. That's right, who knows what could be there. Make sure you understand the default value policy of the language and platform you are using.

17.3 `null` – no memory address

Remember how we talked about how array variables store the memory address to the array? Strings (and all objects) actually work the same way. That is, your string variable only stores the address of where the string is in memory. Again, nice Processing syntax hides this from you, but the string itself is stored off in memory the same as an array.



What if the variable does not yet have an address in it? **Computers have a special value which means no address. In computer science, this is called `null`: no memory address.**

For example:

```
String[] names = new String[10];  
println(names[0]);
```

your output is `null`. Try it.

Processing has a special keyword, `null`, that you can use to refer to this special value. For example, you can check if an array bin in a string array is empty by

comparing it to `null`:

```
String[] names = new String[10];
if (names[0] == null)
    println("bin 0 is empty");
```

But! You may be thinking that we should never use `==` with strings. We'll come back to this later in this unit.



Another point: ***null is not the same as the empty string!*** This is confusing as heck, but an empty string is still a string, with length 0. `null` is not a string at all, and if you try to check its length, the program crashes.

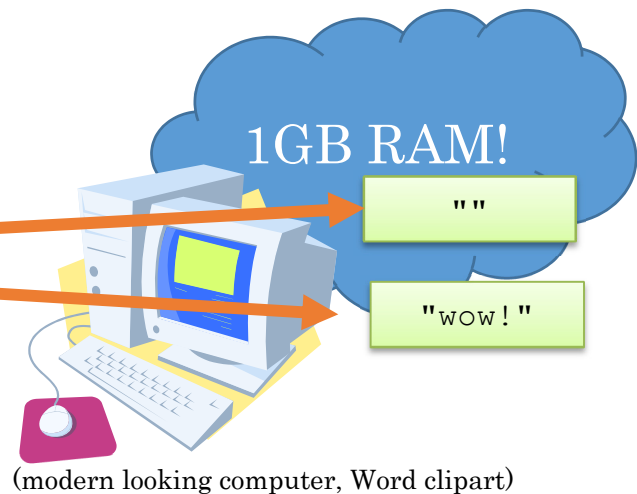
```
String[] names = new String[10];
println(names[0].length());
```

You get a problem called "null pointer exception." Means there is no string to check the length of.

Look at the following code and visual that shows how the three different strings are stored in memory:

```
String s1 = null;
String s2 = "";
String s3 = "wow!";
```

`s1` doesn't point anywhere, since it is set to `null` and does not have an address. Both `s2` and `s3` point to their own respective strings in memory. The empty string is still a string, the same as "wow!", it just has length 0.



Finally, `null` cannot be used for primitives. You cannot set an integer or float variable to `null`, since these variables store your actual data and not a memory address.

17.4 Copying Arrays

How do you make a copy of a primitive variable like an integer or floating point? This should be obvious:

```
int i = 1982;
int j = i; // copy i into j, j now 1982
i = 1999; // replace i
```

```
println(i + " " + j); // what is the output?
```

The output is, as you'd expect:

```
1999 1982
```

`i` has the new value, 1999. `j` remembers the copy it took from `i` earlier, 1982. This works as you should expect, no surprises here. Let's do something similar with an array

```
int[] i = {1, 2, 3};  
int[] j = i; // copy i into j..?  
i[0] = 1999; // change i only  
println(i);  
println(j);
```

What is the output?

```
[0] 1999  
[1] 2  
[2] 3  
[0] 1999  
[1] 2  
[2] 3
```

The top array is `i`, and the bottom is `j`. `i` originally had 1 in bin 0. We copied `i` to `j`, hopefully preserving that old value, so we expect that `j` is the old `i`. Then we modify the `i` array to store 1999 in the bin. However, **both** arrays are updated!! This is confusing, how can we update two arrays just with one command?

In order to understand what happened we need to think again about arrays and memory. First, let's create `i`. We have a variable, and create a new array off in memory to store the 1, 2, 3. The variable points to the memory address.

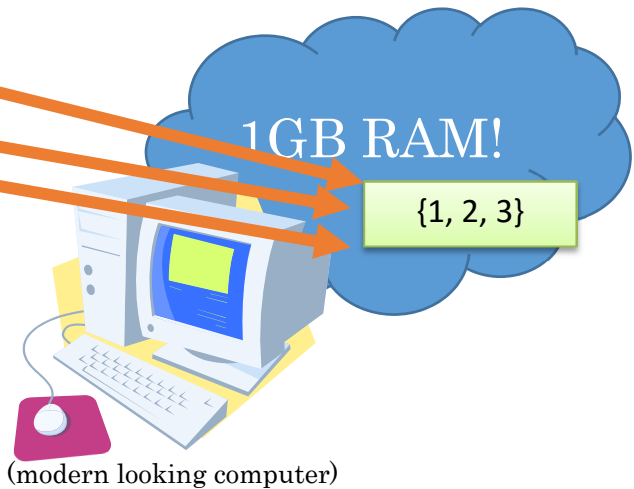
When we create the array variable for `j`, it doesn't point anywhere at first. When we set it to `i`, Processing does exactly what it does for all variables: copy the values! Unfortunately for us `i` does not hold the array, but rather, holds the memory address of the array. Therefore, Processing copies the memory address from `i` to `j`, and `j` gets the same memory address.

Consider the following graphic and code:

```
int[] i = {1, 2, 3};
int[] j = i;
i[0] = 1999;
println(i);
println(j);
```

When `i` is created, we get a new array in memory. When `i` is copied into `j`, however, we just copy the address, so both `i` and `j` have the same address, they both point to the same array in memory. When we set `i[0]` to 1999, it gets the address from `i`, goes off to memory, and changes that single array.

As such, both the `i` and `j` arrays appear to be changed.



This didn't copy the array at all, just copied the memory address from one variable to another. We now have two variables that work on the same piece of memory, the same array.

What we wanted was this: (not working code)

```
int[] i = {1, 2, 3};
int[] j = . ? somehow copy i, and save into j..?
```

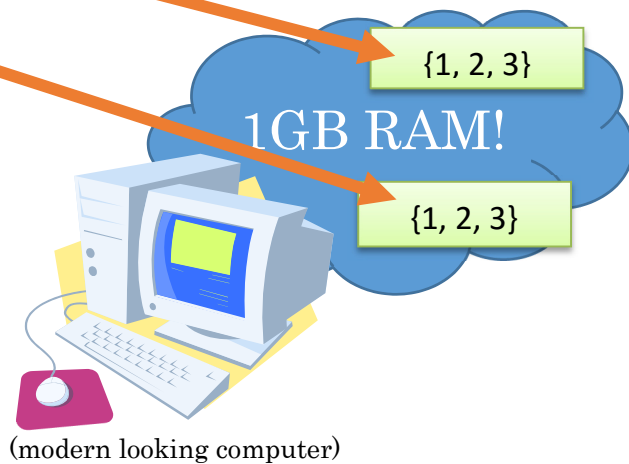
We want an *actual* copy in memory, for Processing to create a new array, so that we can modify one and the other doesn't change.

In this visual, if we change the first bin in the array pointed to by `j`, then this doesn't impact `i`. Our code above doesn't work, but how can we achieve this?

This requires two steps:

- ♦ Create a new array with the same size.
- ♦ Copy the data over from the old array to the new one.

That is, we have to do it manually.



We can create a new array using the syntax we already know. Then, we can copy data over from the old array into the new one using a `for` loop.

```
int[] i = {1, 2, 3};
int[] j = new int[i.length]; // new array same size as i

// copy i into j
for (int bin = 0 ; bin < i.length; bin++)
    j[bin] = i[bin]; // bin-by-bin copy

i[0] = 1999; // only i is modified. j is different array
println(i);
println(j);
```

17.5 Comparing Arrays

Comparing arrays introduces similar problems as copying arrays. Our traditional mechanisms do not make intuitive sense unless we think about arrays and memory. Look at the following example:

```
int[] i = {1,2,3};
int[] j= {1,2,3};
println(i==j);
```

What do you expect will happen? We can see that the arrays have the same data stored in them, so if we use `==` to compare them, we would expect to get `true`. However, if we run this, the result is `false`.

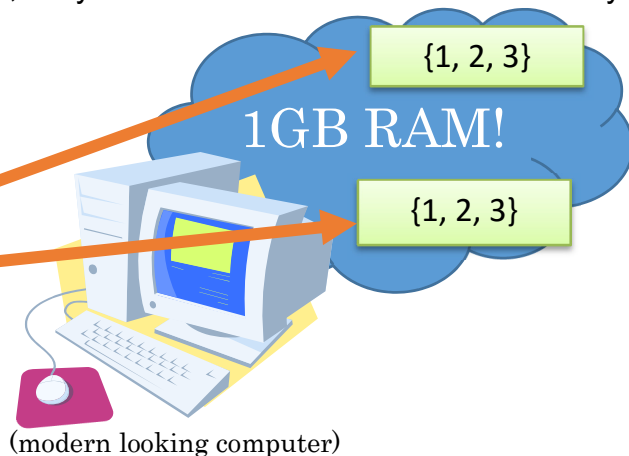


The reason for this is that the **`==` compares what is in the variables: the memory addresses**. Since we have two arrays, they are at different locations in memory. When we compare them, of course they are false.

Consider the same code again but with the visual aid:

```
int[] i = {1,2,3};
int[] j= {1,2,3};
println(i==j);
```

Here, you can clearly see that, although the `i` and `j` arrays store the same data, they have different



addresses. Therefore, the result of the `==` operation is false.



In fact, **this is why you cannot use `==` for strings**. It just compares the memory addresses of where the strings are stored, which is generally not what we want. If we want to see if two strings are equal to each other, we actually need to compare string down to the characters themselves.

So, when we compare arrays, what we really want is a deep comparison: we need to look at each bin individually and see if they are equal across the arrays. We can do this with a `for` loop, and the following algorithm:

- ♦ Assume the arrays `a`, `b` are equal
- ♦ Go through each bin `i` in the array with a `for` loop
 - if `a[i]` does not equal `b[i]`, then the array isn't equal

In code:

```
// assume we have two int arrays, a and b, of the same length
boolean equals = true;
for (int i = 0; i < a.length; i++)
{
    if (a[i] != b[i])
        equals = false;
}
```

At the end of this, if there is any bin `i` where the arrays do not match, then the arrays are considered to be not equal.

What if the array is very big, and, we find out very early (in the first bin!!) that the arrays are not equal? The above code will keep on trucking and continue checking the entire array – a complete waste of work. How can we quit the `for` loop as soon as we find out they are not equal? We can modify the boolean test in the loop to continue also while they are still considered to be equal:

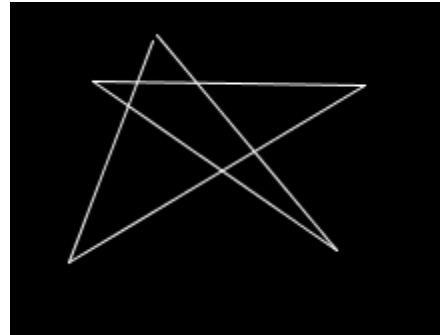
```
// assume we have two int arrays, a and b, of the same length
boolean equals = true;
for (int i = 0; i < a.length && equals; i++)
{
    if (a[i] != b[i])
        equals = false;
}
```

The `for` loop will quit when either of the conditions are false: not equals, or, `i` is not less than length.

With strings, we do not need to do this work because someone has already written the code for us. We get to use that function using the method `.equals` on the strings involved. However, under the hood, they do a very similar operation.

17.6 Example: wandering star!

Let's make a program where we have a 5 pointed star, with its points wandering around (moving randomly). If the mouse is clicked, return them to their original locations.



We will do this by making two arrays for the points of the star, one for `x` and one for `y`. In the draw, we use a `for` loop to draw lines between all the points, and, move each point randomly. When the mouse is clicked, we want to return to the original points – but how do we do this? As we modify the points to move randomly, then we lose the original positions! We need to copy them, to back them up.

We need two sets of arrays

- ♦ The original points
- ♦ A copy of the points, that we modify to make wander around. We can change this copy without changing the original.

First let's make the original points. I will give you the data:

```
float[] xOriginal = {100, 140, 40, 160, 60};
float[] yOriginal = {40, 160, 80, 80, 160};
```

Note that these two arrays are the same size. At bin 0, we have the `x`, `y` for point 0, at bin 1 we have the `x`, `y` for point 1, and so on.

Then, let's setup our `draw` loop to draw them. Note that we need five lines, and use modulo to wrap around the last point to 0.

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

Add code to make each point move by a random amount up to some maximum. I

have my max set to 3 in a variable called `WANDER`. Make sure to do this inside the for loop, for each point. You could use a separate `for` loop for this, but we'll combine for now to save work:

```
for (int i = 0; i < xOriginal.length; i++)
{
    int next = (i+1)%xOriginal.length;
    line(xOriginal[i], yOriginal[i],
        xOriginal[next], yOriginal[next]);
    xOriginal[i] += random(2*WANDER) - WANDER;
    yOriginal[i] += random(2*WANDER) - WANDER;
}
```

If you make sure to setup properly, clear the background, and set the stroke color, you should see a 5-pointed star that wanders around.

However, we need to add the functionality of setting the points back to their original. In our current version, we modify the original points so they are lost. What we need to do, instead, is to first make a copy of the points, and work on those. That way, we still have the original ones for use later.

Let's add the two new arrays and instantiate them with the proper size, at the top of the program:

```
float[] xPoints = new float[xOriginal.length];
float[] yPoints = new float[yOriginal.length];
```

This will serve as our copies.

First, create a void function that copies those original points, verbatim, into the new arrays. This will make the new arrays have the exact same points as the originals. Since we already instantiated the copy arrays, all we need to do is to copy all the data from the bins straight over, like we did in the previous section.

```
void copyOriginal()
{
    for (int i = 0; i < xPoints.length; i++)
    {
        xPoints[i] = xOriginal[i];
        yPoints[i] = yOriginal[i];
    }
}
```

We can now use this to copy the original points into our new arrays. Let's first do this on program startup, and also when the mouse is pressed.

The next piece of the puzzle is that the `draw` block needs to be updated so that, instead of using the original arrays, it uses the copies. This way, we are drawing and moving the copied points, while not touching the originals. When the mouse is pressed, we re-copy the originals into the copies so that it starts over.

Here is the whole program.

```
final float WANDER = 3;

float[] xOriginal = {100, 140, 40, 160, 60};
float[] yOriginal = {40, 160, 80, 80, 160};
float[] xPoints = new float[xOriginal.length];
float[] yPoints = new float[yOriginal.length];

void setup()
{
  size(500, 500);
  copyOriginal();
}

void copyOriginal()
{
  for (int i = 0; i < xPoints.length; i++)
  {
    xPoints[i] = xOriginal[i];
    yPoints[i] = yOriginal[i];
  }
}

void draw()
{
  background(0);
  stroke(255);
  if (mousePressed)
    copyOriginal();

  for (int i = 0; i < xPoints.length; i++)
  {
```

```
int next = (i+1)%xPoints.length;
line(xPoints[i], yPoints[i],
     xPoints[next], yPoints[next]);

xPoints[i] += random(2*WANDER) - WANDER;
yPoints[i] += random(2*WANDER) - WANDER;
}
}
```



Check your Understanding

17.7 Check Your Understanding: Exercises



Exercise 1. Does the following code return true or false?

```
float[] a = {100, 140, 40, 160, 60};
float[] b = {100, 140, 40, 160, 60};
println(a==b);
```



Exercise 2. Using the == operator on arrays and strings does not compare the array or string, but it is still correct code. When would you want to use this? Why is it even allowed?



Exercise 3. What is wrong with the following code?

```
float[] a = {100, 140, 40, 160, 60};
a = null;
println(a[0]);
```

Using your knowledge of arrays and memory, explain what happens.



Exercise 4. Given some array of integers a, make a new array b that is double the size of a, and has the data from a in the first half of its bins (all of a).



Exercise 5. Implement a series of special array comparison techniques that compares two arrays, a and b.

- a. Given two integer arrays, they are considered equal if a) they have the same length, and b) the data in them sum to the same total.
- b. Given one integer array and a floating point array, they are considered equal

- if a) they have the same length, and b) the floating point values cast to integers equal the values in the integer array.
- c. Given one integer array and one boolean array, they are considered equal if
- a) they have the same length, and b) an even number corresponds to true, and an odd number corresponds to false.



Exercise 6. Make a program that generates 10 random points on startup, and stores it in two arrays (one for x and one for y). In the `draw` block, put ellipses at each point in random colors.

- a. If the mouse is pressed, double the number of points, adding new random ones. As the array size changes, it should not lose its data, and, the rest of the code should work properly.
- b. If the keyboard is pressed, halve the number of points, removing every second point.

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.

(page intentionally left blank)