

Part III: Another Way to Build an Array

This discussion is related to class code set 12C. You should have that code open as you read through this.

One nice thing about arrays and 15-102 is that we control the data. We have not needed “outside” data. We have had the luxury of initializing the arrays in our code.

```
int [ ] numbers = { 1, 5, 4 };
```

We call this the initializer list → which contains the values needed in our program. Processing counts the number of values inside the braces and makes the array exactly big enough to contain the data. It then copies the values of the numbers in the list into the array with the beginning value being copied into element [0], the next value into element [1], . . . This works for any type of array.

Let's change the playing field a bit. What if we want random values in the array? We might do this?

```
int [ ] numbers = {int( random( 10 ) ), int( random( 10 ) ),int( random( 10 ) )};
```

This works but if we want an array of 100 random numbers, the code would get out of hand very quickly.

There is another way! We can “new” the array. That's right, we are verbing a perfectly good adjective... sigh... Before we continue, let's look at the syntax:

- First we build the array reference - no array, just the reference:

```
int [ ] numbers;
```

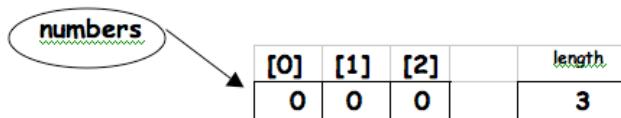
What we have is an “empty” reference. It is said to be **null**. We diagram it like this:



- Next, we build the array for numbers to reference. For this discussion, we want this array to have just three elements but it could have any reasonable number of elements.

```
numbers = new int [ 3 ];
```

Now our diagram looks like the traditional array diagram we have used since the beginning of our work with arrays:



Since the array is a global variable, the elements are initialized to zero.

Here is the Processing program that does this:

```
int [ ] numbers;

void setup( )
{
    size( 300, 300 );
    numbers = new int[ 3 ];
    initArray( numbers );
}
```

We have to write the code for the function `initArray()`. The definition of `initArray()` is below and it looks like code we have been writing for several weeks:

```
void initArray( int [ ] anyArray )
{
    for( int i = 0; i < anyArray.length; i++ )
    {
        anyArray[ i ] = int( random( 100 ) );
    }
}
```

For a discussion of code very similar to the code in `initArray()` or for a review of arrays from the beginning, you should refer back to the board notes and class code for 1001.

The `println()` function actually prints the array and its contents with very nicely formatted output. Below is a slightly modified version of the program above and the output it generates:

```
void setup( )
```

```
{  
    size( 300, 300 );  
    println( "Printing array before we new it it:" );  
    println( numbers );  
    numbers = new int[ 3 ];  
    println( "Printing array before we initialize it:" );  
    println( numbers );  
    initArray( numbers );  
    println( "Printing array after we initialize it:" );  
    println( numbers );  
}  
  
void initArray( int [ ] anyArray )  
{  
    for( int i = 0; i < anyArray.length; i++ )  
    {  
        anyArray[ i ] = int( random( 100 ) );  
    }  
}
```

Here is the output:

```
Printing array before we new it it:  
null  
Printing array before we initialize it:  
[0] 0  
[1] 0  
[2] 0  
Printing array after we initialize it:  
[0] 26  
[1] 32  
[2] 99
```

Here is code very similar to what we wrote in class:

```
final int MAX = 5; // ← This is a constant

int [ ] a;
int [ ] b;
color [ ] col;

void setup( )
{
    size( 300, 300 );

    a = new int [ MAX ];
    b = new int [ MAX ];
    col = new color[ MAX ];

    initializeIntegerArray( a );
    initializeIntegerArray( b );
    initializeColorArray( );

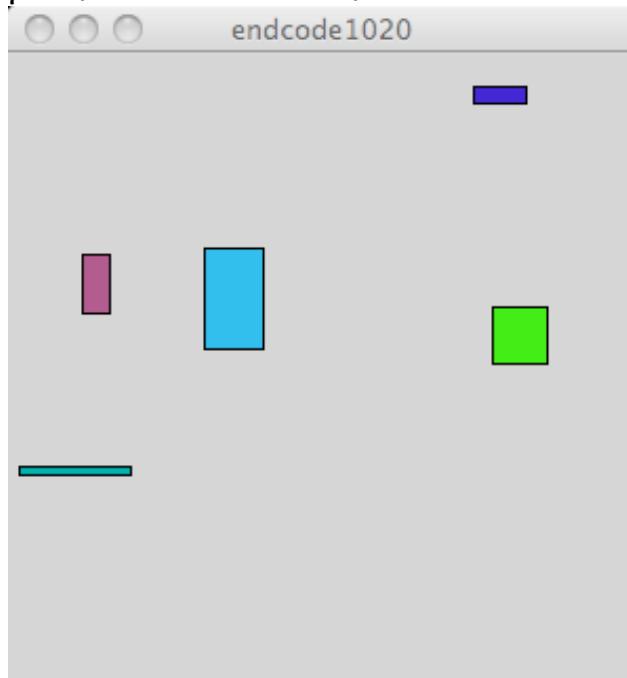
    drawBoxes( );
}

void initializeIntegerArray( int [ ] anyArray )
{
    for( int i = 0; i < anyArray.length; i++ )
    {
        anyArray[ i ] = int( random( width ) );
    }
}

void initializeColorArray( )
{
    for( int i = 0; i < col.length; i++ )
    {
        col[ i ] = color( int( random(255 ) ),
                           int( random(255 ) ),
                           int( random(255 ) ) );
    }
}

void drawBoxes( )
{
    for( int i = 0; i < a.length; i++ )
    {
        fill( col[i] );
        rect( a[i], b[i], random( width/5 ), random( height/5 ) );
    }
}
```

Here is the output from one run of this code:



This code uses the “newing” discussed in the first pages of this part of notes. Several things about some of the code: First, the three calls to initialize the arrays;

```
initializeIntegerArray( a );
initializeIntegerArray( b );
initializeColorArray( );
```

Why do the calls to `initializeIntegerArray()` have an argument while the call to `initializeColorArray()` does not? The answer is that there are two arrays of integer in the program.

- We want to use a single function to initialize both of them. In order to do this, we have to use the **argument binding** we have been talking about since we wrote our first function definition (`drawIntials(int, int, int, int)`). For a review of this and how it works with arrays, you should refer back to the notes for 1006.
- There is only a single array of color so the function `initializeColorArray()` can directly access the array. If there were two or more arrays of color, we would have to use **argument binding** for this function.

One last idea is a concept of **parallel arrays**. The three arrays in this code are said to be or described as “**parallel arrays**.”

There is nothing in the syntax that makes the “parallel.” They are parallel because of the way they are used. We can see this in the code in this function:

```
void drawBoxes( )
{
    for( int i = 0; i < a.length; i++ )
    {
        fill( col[i] );
        rect( a[i], b[i], random( width/5), random(height/5) );
    }
}
```

The arrays are parallel because the **[i]**th elements of all three arrays are used to draw the **[i]**th box. Color **col[0]** is the color of the zeroth box. The zeroth box uses **a[0]** for its **x** coordinate value and **b[0]** for its **y** coordinate value. It is this relationship in the code that makes the arrays parallel.

If you look at the class code set 11 Demo 3, you will find the program of bouncing squares that Jim used to introduce the idea of arrays. The code in the program uses six parallel arrays. Five of the arrays contain **int** values and the sixth contains the **colors**:

```
int [ ] x, y, edge, dX, dY;
color[ ] col;
```

The code in this program is very similar to the code used in these notes.

One last thing in this part: **constants** . . .

We mentioned this one time last week but it is worth repeating it. The line of code in red on page 4 near the top is a **constant**. The “final” makes it a **constant**. This is the syntax for declaring and initializing **finals** in Processing. When we use this “**newing**” technique with arrays, we should set the size of the arrays with a constant. Doing this allows us to alter the array size with a single edit. By convention, constant names are **all uppercase letters**.