Part I: Arrays Basics #2:

This discussion is related to class code set12A. You should have that code open as you read through this.

The first line of the program:

```
int [ ] numbers = { 3, 5, 8, 1, 3, -2, 4, 11, 7, 6 };
```

declares and initializes the array. Building arrays like this must be done globally in one statement or it will not compile.
This line of code builds an array with a length of 10. The field, numbers.length stores the value 10 which is the number of elements in the array or the size of the array.

The pattern we follow for much of the array work involves using a for loop to get to each value stored in the array. This is called *traversing the array*. Unless specified otherwise, we often begin with the element [0] and move to element [length-1]. However, this is a pattern and some requirements (as seen in the fourth example below) require starting and/or stopping at other elements of the array.

The first example in the code prints the array on a single line. Here is the function:

```
void printArray( )
{
    println("Values in the array:");
   for ( int i = 0 ; i < numbers.length ; i++ )
   {
      print( numbers[i] + "  " );
   }
   println( );
}
```

Let's trace the execution of the for loop:

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

Value of i	Evaluation of test: i < numbers.length (*length's value is 10*)	Array element being visited	Value of array element being visited	What the visit does with the value
0	true	[0]	3	Print it
1	true	[1]	5	Print it
2	true	[2]	8	Print it
3	true	[3]	1	Print it
4	true	[4]	3	Print it
5	true	[5]	-2	Print it
6	true	[6]	4	Print it
7	true	[7]	11	Print it
8	true	[8]	7	Print it
9	true	[9]	6	Print it
10	false			

This rather verbose way of tracing the execution of the for loop shows how Processing uses the for loop variable, i to access or "visit" each element of the array. The word visit means that we do something with the value stored in the array. The "something" we do is specified in the problem we are solving. The specification here was to print each element.

The second function in the code has the task of computing and returning the average back to the draw( ) function where it was called so it could be printed.

```
float getAverage( )
{
   float sum = 0;
   for ( int i = 0 ; i < numbers.length ; i++ )
   {
      sum = sum + numbers[i] ;
   }
   return sum/numbers.length;
}
```

This function is not a void function because it must return the average to draw( ) for printing. The reason the function returns a float is because a fractional answer is often a better representation of an average. The average of the values

1 and 2 is best represented by the value 1.5. If we use int values to do this we get a different result:

2 / 1 ➔ 1

Remember the rules of integer division – the result must be iint.

Here is a similar tracing of the for loop:

for (int i = 0 ; i < numbers.length ; i++)

Value of i	Evaluation of test: i < numbers. length	Array element being visited	Value of array element being visited	What the visit does with the value	Value of local variable sum
					0.0
0	true	[0]	3	Add it to sum	3.0
1	true	[1]	5	Add it to sum	8.0
2	true	[2]	8	Add it to sum	16.0
3	true	[3]	1	Add it to sum	17.0
4	true	[4]	3	Add it to sum	20.0
5	true	[5]	-2	Add it to sum	18.2
6	true	[6]	4	Add it to sum	22.0
7	true	[7]	11	Add it to sum	33.0
8	true	[8]	7	Add it to sum	40.0
9	true	[9]	6	Add it to sum	46.0
10	false				

The third example prints values in the array that are greater than the average:

```
void  printValuesGreaterThanAverage( float average )
{
   println("Values greater than the average of " + average + ": ");
   for ( int i = 0 ; i < numbers.length ; i++ )
   {
     if ( numbers[i] > average)
    {
       print( numbers[i] + "  " );
    }
   }
   println( );
}
```

Here is a another trace of the for loop:

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

Value of i	Evaluation of test: i < numbers. length	Array element being visited	Value of array element being visited	Evaluation of test: numbers[i] > average The average is 4.6	What this visit does with value
0	true	[0]	3	false	nothing
1	true	[1]	5	true	print it
2	true	[2]	8	true	print it
3	true	[3]	1	false	nothing
4	true	[4]	3	false	nothing
5	true	[5]	-2	false	nothing
6	true	[6]	4	false	nothing
7	true	[7]	11	true	print it
8	true	[8]	7	true	print it
9	true	[9]	6	true	print it
10	false				

The fourth example uses a different pattern in the for loop. The for loop must begin at element [1] and stop at element [numbers.length-1]. The reason is the task specified in the name of the function:

printValuesGreaterThanBothNeighbors( )

To understand the task we have to define the term neighbor. For this task a neighbor is the element that is either immediately before and after an array element. For element [3], the neighbors are elements [2] and [4]. Note that not all elements have two neighbors. Element [0] and element [length-1] have only one neighbor(elements [1] and [length-2] respectively). This function must print only those elements that have values greater than both neighbors. Given the above, lets think about this… In order to "qualify" for printing, an element must have two neighbors. Elements [0] and [numbers.length] do not have two neighbors. If our code tries to visit the element before element [0] or after element [numbers.length], the program will crash. So we have to alter the pattern of the for loop. Here is the function definition:

```
void printValuesGreaterThanBothNeighbors( )
 {
   println("Values greater than both neighbors: ");
   for ( int i = 1 ; i < numbers.length-1 ; i++ )
   {
     if ( numbers[i] > numbers[i-1] &&
          numbers[i] > numbers[i+1])
    {
       print( numbers[i] + "  " );
    }
   }
   println( );
 }
```

For practice, we will let you trace the execution of the for loop in this code:

Here is the array again so you do not have to flip back and forth:

```
int [ ] numbers = { 3, 5, 8, 1, 3, -2, 4, 11, 7, 6 };
```

```
for ( int i = 1 ; i < numbers.length-1 ; i++ )
```

Value of i	Evaluation of test: i < numbers. length	Array element being visited	Value of array element being visited	Evaluation of test: numbers[i] > numbers[i-1] && numbers[i] > numbers[i+1]	What this visit does with value

Part II Arrays as arguments and return types:

This discussion is related to class code Set12B. You should have that code open as you read through this.

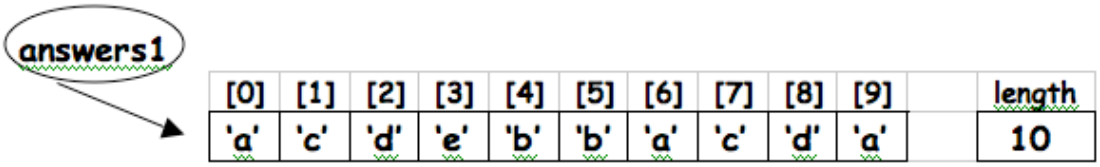
This set of notes focuses on using arrays as arguments and as return types. The last part is code for a type of search called a filter.

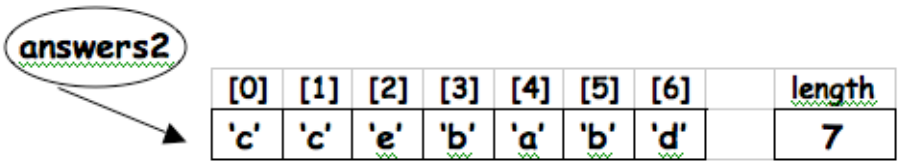
You should read over the board notes bn14PartA and revisit the first part of these notes if you are still foggy on how to declare and initialize an array and how to use a for loop to traverse the array and what we can do when we visit each element of the array. These notes assume you know what those terms mean.

We begin with two arrays of char:

```
char [ ] answers1 = { 'a', 'c', 'd', 'e', 'b', 'b', 'a', 'c', 'd', 'a' };
char [ ] answers2 = { 'c', 'c', 'e', 'b', 'a', 'b', 'd' };
```

Here is how we would draw Processing's view of these arrays:





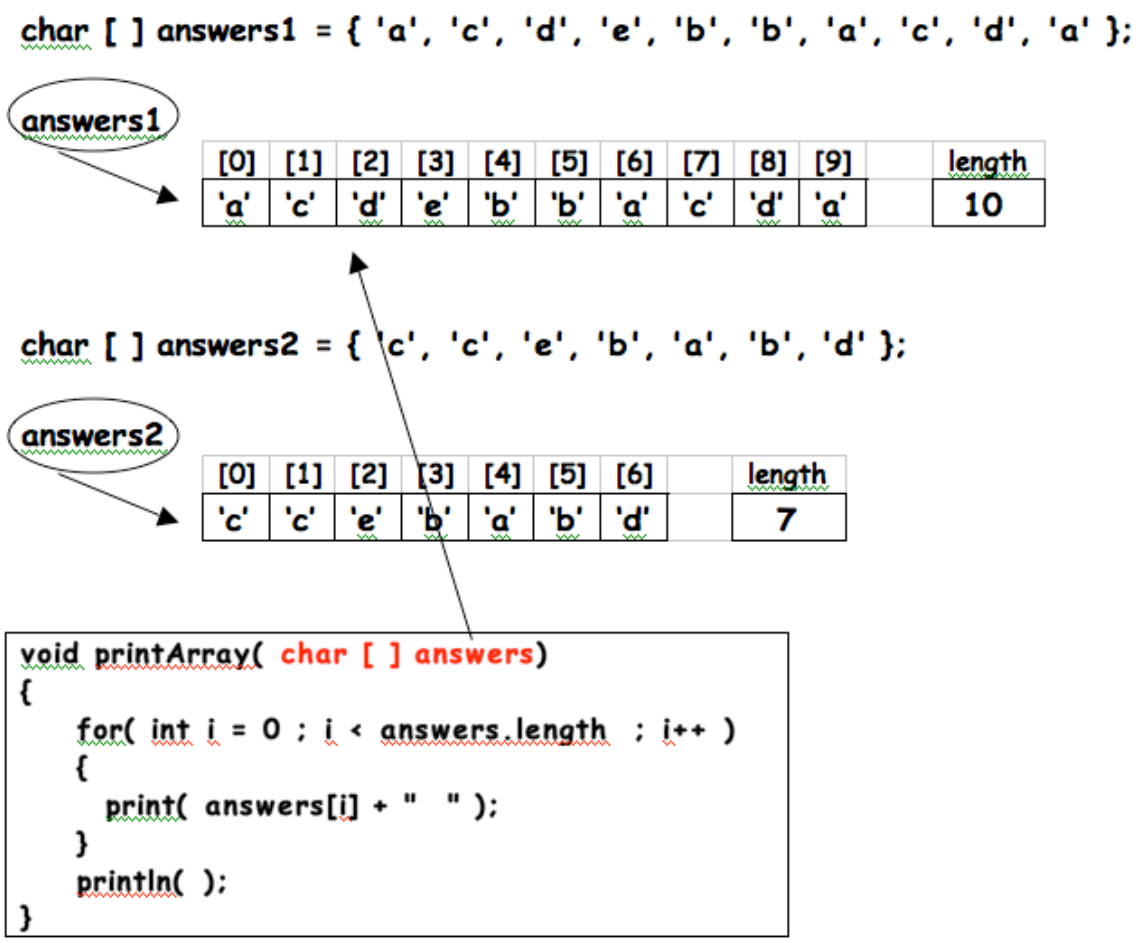
The first two function calls print each array:

```
printArray( answers1 );
printArray( answers2 );
```

Processing does not make copies of the data for the arguments in the definition as it would for primitive variables. What Processing does is make copies the arrow that connects the reference to the array. This sounds very confusing so let's make another drawing.

Here is how the program looks for this function call:

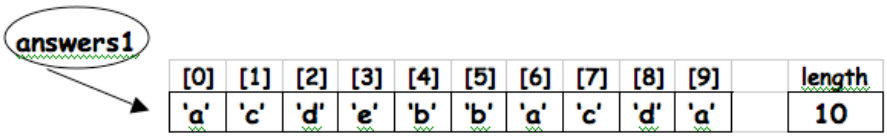
```
printArray( answers1 );
```



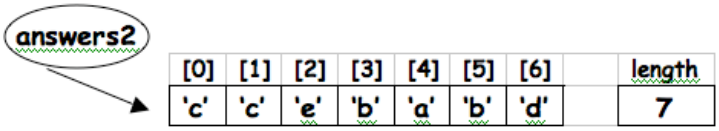
When printArray is called with answers1 as the argument, the argument answers in the definition is assigned to reference the same array that answers1 references. Using this diagram, when the function printArray ( ) needs to know what the length of the array is, it "follows" the reference arrow to the array and looks up the length – which is 10. When the function is visiting an element and needs to know the value of an element, it does the same thing.

A similar picture forms for the second function call:
Here is how the program looks for this function call:
printArray(answers2);

```
char [ ] answers1 = { 'a', 'c', 'd', 'e', 'b', 'b', 'a', 'c', 'd', 'a' };
```




```
char [ ] answers2 = { 'c', 'c', 'e', 'b', 'a', 'b', 'd' };
```




```
void printArray( char [ ] answers)
{
    for( int i = 0 ; i < answers.length  ; i++ )
    {
      print( answers[i] + "  " );
    }
    println( );
}
```

For the second call where the argument is answers2, Processing assigns the argument, answers in the definition to reference the same array as answers2. When it needs to know the length of the array, it follows the arrow to the array and finds the length of this array to be 7.

Some differences between arrays and primitive variables:

- **Arrays can have multiple values – primitive variables can have only one value.**
- **Arrays can have multiple references to them – primitive variables can have only one name.**

If you are not sure what the for loop is doing or how it is working in this code, you must refer back to the previous set of class code or refer to Shiffman. The execution of the for loop with the array is explained and traced in detail.

Do not go on with this set of notes if you do not understand how the loop and the array work together. You will be wasting your time.

Next we have this line of code:

```
char [ ] answersAll = concat( answers1, answers2 );
```

The left side of the assignment operator is building a new array reference. There is no array – just a reference. The actual array is built and returned by the function concat( ). The concat( ) function is part of the Processing API. The concat( ) function builds a new array that contains the elements of the two arrays in the argument list. The two arrays in the arguments list (answers1, answers2) are not modified or destroyed. Their values are copied into the elements of the new array. A reference to this new array is returned to the right side of the assignment operator where it is assigned to the array reference, answersAll.

There are a number of functions in the Processing API that return references to new arrays. Some of these might be useful to you in the last half of the semester. You should explore these to see what they do and how they work.

The next line of code is :

```
char [ ] answersOdd = buildAnswersOdd( answersAll );
```

Here again, the left side of the assignment operator builds a new array and returns a reference to it to be assigned to answersOdd. Unlike the previous line, there is no definition of buildAnswersOdd( )in the Processing API. We have to define it.

Here is the definition of buildAnswersOdd( )

```
char [ ]  buildAnswersOdd( char [ ] answers )
{
    char [ ] temp = { };
    for( int i = 1 ;i < answers.length  ;  i = i + 2 )
    {
       {
         temp = append( temp, answers[i] );
       }
    }
    return temp;
};
```

Here is a look at the parts of this definition:

char [ ] This is the return type. OK – why is this function returning an array of char. The answer is not a guess or the result of some form of mystical reasoning. The answer is in the line of code where the function is called:

```
char [ ] answersOdd = buildAnswersOdd( answersAll );
```

Read this from right to left. It reads as:

"*buildAnswersOdd( )will return something to be assigned to answersOdd*."
The next question is to ask is, "what is answersOdd?"
We continue to read from right to left:

```
char [ ] answersOdd
```

***T*his reads as, "*answersOdd is an array of char.*"**

This tells us that the array buildAnswersOdd( )must return an array of char.

```
char [ ] temp = { };
```

If we are going to return an array of char, we must first build one. The reference temp is a local variable. Processing does not initialize local variables so we have to. This line of code initializes the array reference temp to an array of char that is empty. Its length at this point in the execution is zero.

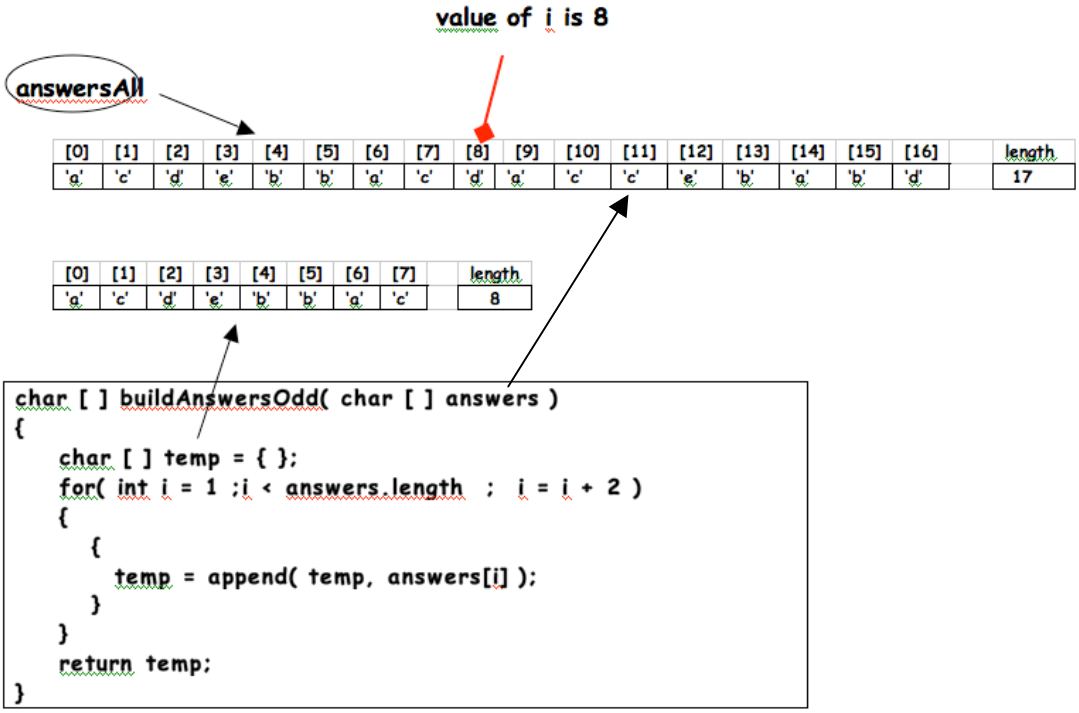
```
for( int i = 1 ;i < answers.length  ;  i = i + 2 )
```

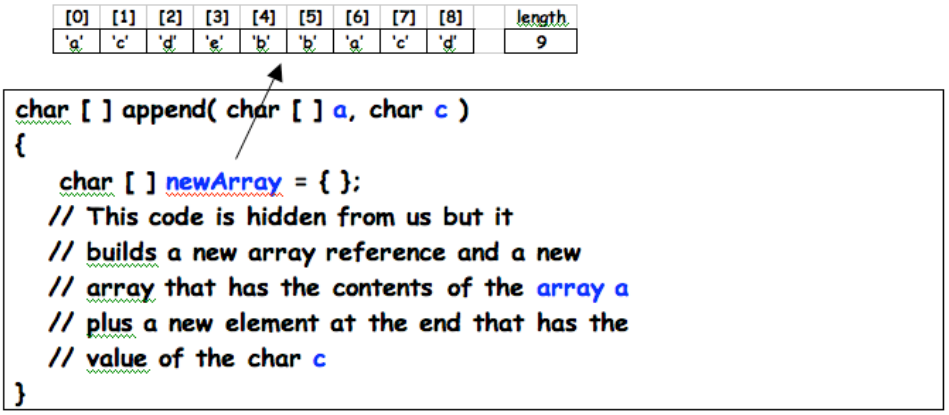
The function must build a new array containing the elements of the odd indexes in the array referenced by the argument. We can do this in different ways. Two were done in class. In this example the loop starts at element [1] instead of element [0]. It traverses and visits <u>every other element</u> or the odd elements. Thi happens because of the way the loop increments the variable i. Instead of i++ which is the "usual" pattern, i is incremented by 2 with this code: i = i + 2

`temp = append( temp, answers[i] );` **The append( ) function is in the Processing API. append( ) returns a reference to a new array that contains all of the elements in the array temp plus one new element at the end which has the value of the element in answers[i]. (Read this over several times). The reference to the new array is assigned to temp. temp's old reference to the old array is lost forever. Since there are no references to the old array, it too, is lost forever.**

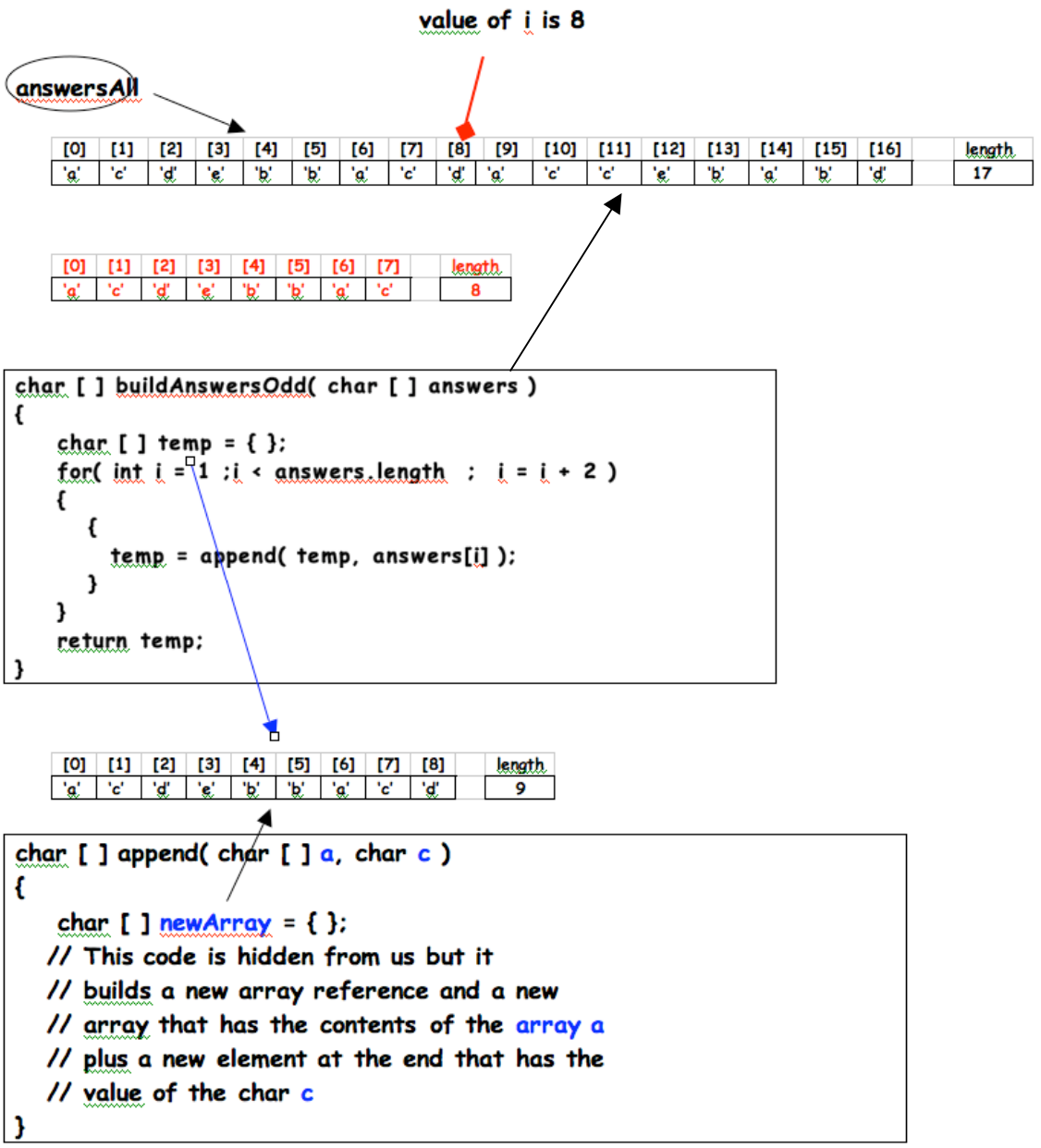
The next page shows the structure of the arrays when the value of the loop variable is 8 BUT before the reference is returned:

Remember, this is before the append( ) function returns the reference to the new array:





The next page shows the structure of the arrays after the append( ) function has returned the reference to the new array:



The array reference temp is no longer pointing to the 8 element array now shown in red. It is pointing to the new array that contains all of the elements in the old array plus one more – the value of element [8] in the array referenced by answers. The old array now shown in red that temp used to reference now has no references. We can no longer access any of the values stored in it. It is lost to us forever.[1]

[1] In the "old" days of programming, this was called a memory leak. If this occurred too often, the program could crash because it ran out of available memory. Processing runs a

This is somewhat complex. It is explained here to show you that a reasonable set of events do occur when we work with arrays. The more of this you understand, the easier your work with arrays will be in the coming weeks.

The last set of function calls:

```
println( "The number of a answers in answers1 is "    + countLetter( answers1,    'a' ) );
println( "The number of b answers in answers2 is "    + countLetter( answers2,    'b' ) );
println( "The number of c answers in answersOdd is " + countLetter( answersOdd, 'c' ) );
```

call a function that demonstrates one form of array search called a filter. An array search occurs when we traverse an array looking for something specific in the array. There are two general types of array searches:

1. **We are looking for one specific value or the first occurrence of a specific value in the array. It may or may not be there. If, and when we find it, we can stop looking. This search is similar to you looking for your keys or your id card in your room when you have misplaced it. You stop looking when you find it**
2. **We are looking for all occurrences of a specific value in an array. We must traverse the entire array and check every element. We cannot stop when we find the first value. This is similar to your searching for dirty laundry before heading down to the laundry room. You do not stop when you find the first pair of dirty socks. You have to look everywhere in your room.**

The second search is often called a filter. We are filtering the array picking out certain values.

The function countLetter( )must traverse the array and count the number of times a specific character is in the array. It is filtering the characters in the array looking or a specific character.

Here is the code:

```
int countLetter( char [ ] answers, char letterToCount )
{
    int letterCount = 0;
    for( int i = 0 ; i < answers.length  ; i++ )
    {
      if ( answers[i] == letterToCount )
      {
         letterCount++;
```

---

program called the garbage collector (yes, that is what it is called) that goes around collecting up unreferenced arrays and returns the memory to the operating system so our program will not run out of memory.

```
      }
    }
    return letterCount;
}
```

countLetter( ) must return a count of a specific character. Since it is counting characters, the type of the count should be int. We should not find half of an 'x'.

`int letterCount = 0;` **We declare a variable to store the count and initialize it to zero.**

`for( int i = 0 ; i < answers.length ; i++ )` **We use the for loop to traverse the entire array since the character we are counting might be in any element.**

`if ( answers[i] == letterToCount )` **We visit each element of the array and "ask" if it is the letter we are counting. To ask the question, we use the if control structure. If the [i]th element is equal to the letter we are looking for, the expression evaluates to true.**

`letterCount++;` **If the expression is true, we increment the variable we are using to store the count. If it is false, we do nothing.**

`return letterCount;` **Finally we return the count of the character we are looking for back to the call where it is printed on the screen.**

Final Thoughts on this part:

There is a lot here – a lot. Arrays are the basis of the second exam. You need to work through this and the code of these same days. If you do not understand any part of all of this, please come to office hours held by Jim or the CAs and do this soon. Week 9 or 10 is too late.

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) ) };
```

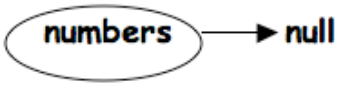
This works but if we want an array of 100 random numbers, the code would get out of hand very quickly.

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;
  ```

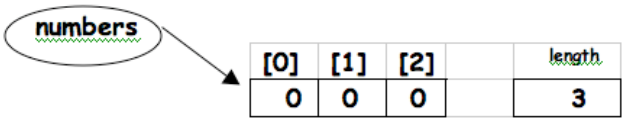
  What we have is an "empty" reference. It is said to be null. We diagram it like this:

- 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 ) );
  }
}
```

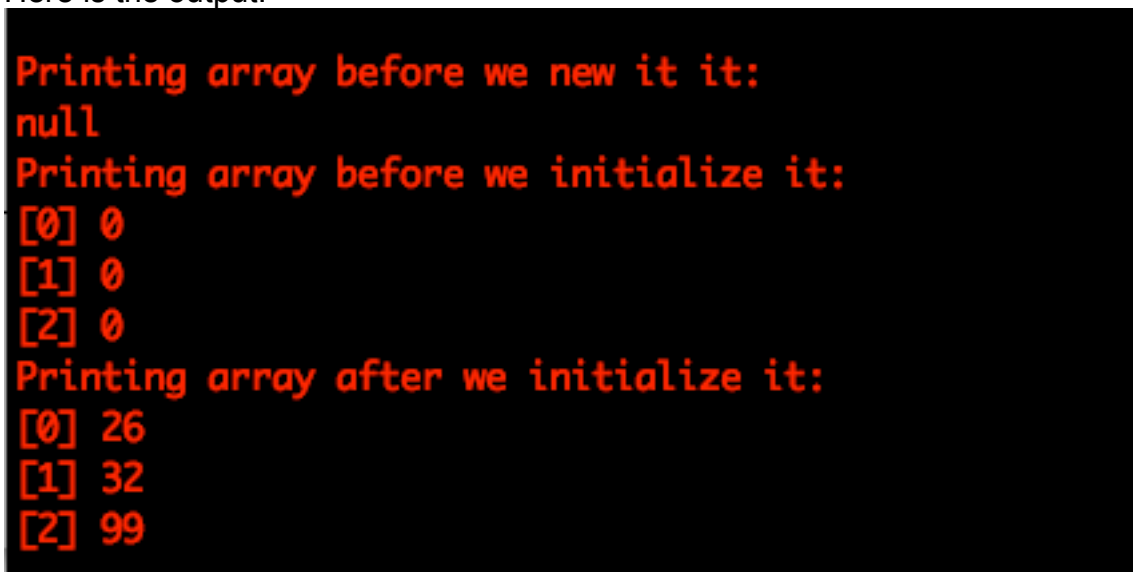
For a discussion of code very similar to the code in initArray() or for a review of arrays from the beginning, you should refer back to the board notes and class code for 1001.

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 ) );
  }
}
```

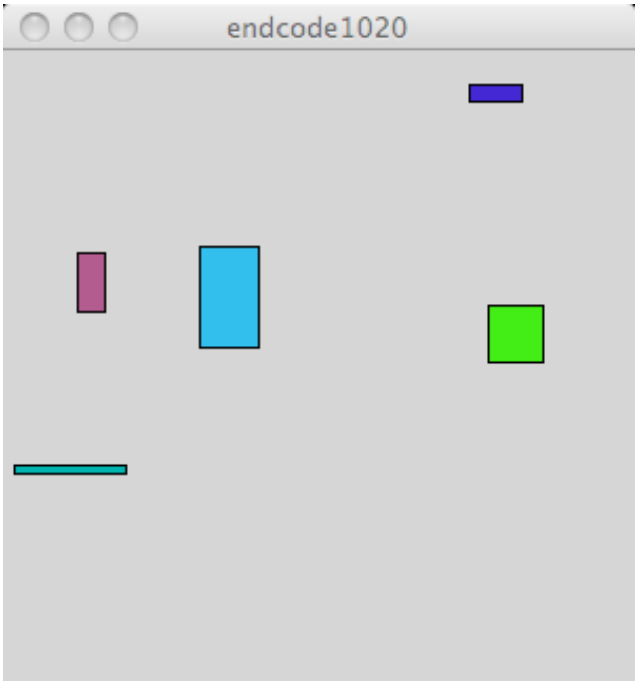
Here is the output:



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 in 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 allow us to alter the array size with a single edit. By convention, constant names are all uppercase letters.