Week 6 Description

This week, we'll look at a variation on the for loop that's specifically intended for use with arrays - it's called the for each loop. We will be learning parallel arrays, how to store fewer object in arrays, counting number of elements in an array that satisfy a predicate.

Next, look how to copy arrays, reverse elements and split arrays. We'll see how Java permits to define a method in terms of itself.

For each loop

There is another type of loop in Java that is made specifically for use with arrays, the **for each loop**. It's easier to use than a regular for loop because it doesn't involve indexes or square brackets. Here's an example:

```
String[] flavourArray = new String[3];
flavourArray[0] = "chocolate";
flavourArray[1] = "strawberry";
flavourArray[2] = "vanilla";

for (String flavour : flavourArray)
System.out.println(flavour);
```

The for each produces the same results as a for loop or a while loop that loops over each element of the array. It **automatically** loops once for each element in the array.

In the for each loop, the **String flavour** portion declares a String variable called flavour. In each iteration, flavour is assigned the value of the next element of the array. The first time through, it is chocolate, the second time it is strawberry, etc. The **: flavourArray** portion tells the system the name of the array.

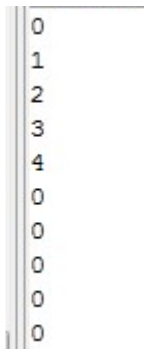
Array details

Storing fewer objects in an array than the array can hold

What happens if we go back to our first example of filling an array with the numbers 0 to 9, but we stop filling at 4 instead of going all the way to the end?

```
       int[] values = new int[10];
        //fill the array with the numbers 0 to 4 using a for loop
        //stop at 5 instead of going to values.length
        for (int i = 0; i < 5; i++)
        {
             values[i] = i;
        }
        //print the numbers in the array using a for loop
        for (int i = 0; i < values.length; i++)
        {
              System.out.println(values[i]);
        }
```

You should know the answer without looking! Since arrays are objects, and objects are initialized by Java, the unassigned elements will contain 0:



```
0
1
2
3
4
0
0
0
0
0
```

Reading data until a sentinel value

The following is an example to read data until sentinel value. Here, we only loop to get data from the user until they enter the number 0. This is a sentinel value.

I've used a do while loop here since we don't know how many values the user will enter.

Note that if they enter more than 10, the program will crash since the array isn't big enough. We could add another condition on the while to test for this and avoid that problem if we wanted to.

```
        int intArray[] = new int[10];
        int number, counter = 0;

        System.out.println("Enter some numbers, enter '0' to quit");
        do {
            number = scanner.nextInt();
            if (number == 0) {
                break;
            }
            intArray[counter] = number;
            counter++;
        } while (number != 0 && counter < intArray.length);

        //print partially filled array
        for(int i = 0; i < counter; i++){
            System.out.println("Element " + (i + 1) + ": " + intArray[i]);
        }
```

Note that in the code above, in order to print only the values actually entered, we save the values when the loop finishes into a variable called counter. We then use this variable as the endpoint in our print loop.

Counting the number of elements in an array which satisfy a predicate

The last common thing we'll discuss about arrays is how to count the number of elements in an array that have a particular value (predicate).

```
//count number of elements that match a predicate (value)
//declare and initialise array
int[] values = {1, 2, 3, 4, 1, 2, 5, 7, 9, 10, 1, 4};
//declare counter for the number of times the value we are looking for occurs
int countValue = 0;
//get value the user wants to count
int searchValue = Integer.parseInt(JOptionPane.showInputDialog(
            "Enter value you want to count: "));
for (int value = 0; value < values.length; value++) {
    if (values[value] == searchValue) {
        countValue++;
    }
}
System.out.println("The value " + searchValue + " occurs " + countValue +
        " times in the array");
```

Here, we create an array and put some values in it. We then declare a variable that will hold the count of times a particular value occurs in the array. We get the value to count from the user, then loop through the array. Each time through the loop, we compare the element to the value we are looking for. If it is equal, we add to our counter. When we're done, the counter tells us how many times the value occurred.

Parallel arrays

An array can only hold one type of data: integers, doubles, objects of the Employee class, etc. This can be inconvenient when you have more than one type of data, and perhaps don't have an object to deal with several types of data.

Here we can use parallel arrays. Parallel arrays are separate arrays where we use the same index in both arrays to refer logically to the same item. We often process parallel arrays in our code at the same time using one loop.

Here is a website with simple examples:
http://mathbits.com/MathBits/Java/arrays/ParallelArrays.htm

Arithmetic with array variables

You cannot do arithmetic with array **variables**:

```
int[] intArray = {2, 4, 6};
intArray = intArray + 2;    // compile time error – can't add int[] + int
```

nor

```
int[] array1 = {2, 4, 6};
int[] array2 = {1, 3, 5};
int[] array3 = array1 + array2;   // compile time error – can't add int[] + int[]
```

You can, of course, do arithmetic on array **elements**, if they are of a primitive type:

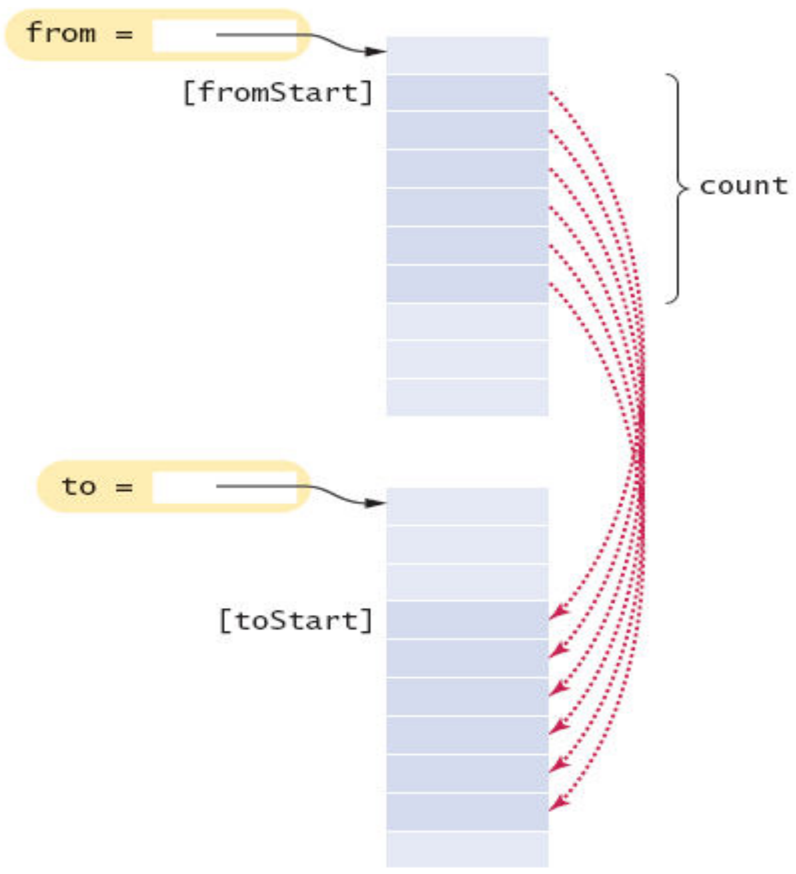
```
int[] intArray = {2, 4, 6};
for (int j = 0; j < intArray.length; ++j)
    intArray[j] = intArray[j] + 2;
```

or

```
int[] array1 = {2, 4, 6};
```

```
int[] array2 = {1, 3, 5};
int[] array3 = new int[array1.length];
for (int j = 0; j < array1.length; ++j)
   array3[j] = array1[j] + array2[j];
```

Library arraycopy method



The System.arraycopy Method

An array variable can refer to an array object of any size. However, an array object, once created, cannot change its size. If you create an array object with length 4, and, at run time, you realize you need more space, there is no way to expand the size of this array. You would have to create a new, larger array, then copy the contents of the old array to the new one.

Because we frequently need to copy arrays, there is a Java API library method to do this. This method is called arraycopy, and is found in the API class System. arraycopy has the format:

```
System.arraycopy(from, fromStart, to, toStart, count);
```

This produces results as shown in the figure above.

Here's a snippet that uses this method:

```
  // declare an array variable, create an array object
```

```
      int[] array1 = {2, 4, 6, 8, 10};

      // print array1
      for (int n = 0; n < array1.length; ++n)
         System.out.print(array1[n] + " ");
       System.out.println();

      /* Now we decide we want to add more data to this array, but there's
no room. Create a new, larger array object and assign it to a temporary
variable. */
      int[] temp = new int[2 * array1.length];

      // copy the contents of the old array to the new one
      System.arraycopy(array1, 0, temp, 0, array1.length);

      /* assign the new array object to the old variable.
      This turns the old object into garbage, but we no longer need it */
      array1 = temp;

      /* Print array1 again, to show that it holds the same data but is now
longer. */
      for (int n = 0; n < array1.length; ++n)
         System.out.print(array1[n] + " ");
      System.out.println();
```

This prints out:

```
2 4 6 8 10
2 4 6 8 10 0 0 0 0 0
```

The arraycopy method takes five arguments. The first argument is the array that you want to copy from and the third is the array you want to copy to. You use array variables for these arguments, but you must have already created array objects and assigned them to these variables. The method does not create a new array object.

In the example above, we were copying the entire contents of the first array, but this method can also be used to copy part of an array. The second argument says where to start copying **from** (the index of the first element that you copy from, in the old array), the fourth argument says where to start copying **to** (the index of the first element that you copy to, in the new array), and the fifth argument says how many elements to copy.

In the example above, we called:

```
System.arraycopy(array1, 0, temp, 0, array1.length);
```

This says to start copying from the element at index 0 in array1, to start copying to the element at index 0 in temp, and to copy array1.length elements (i.e. the entire contents of array1). As above, this gave the output:

```
2 4 6 8 10
2 4 6 8 10 0 0 0 0 0
```

If we had instead called:

```
System.arraycopy(array1, 1, temp, 6, 3);
```

we would have started copying from array1[1], started copying to temp[6], and copied 3 elements. Then the output would have been:

```
2 4 6 8 10
0 0 0 0 0 0 4 6 8 0
```

Comparing Arrays

Comparing Arrays

You can compare two array variables using == (or !=) but this tests whether both **variables** refer to the same object. It does not test whether the **contents** of two different objects are the same.

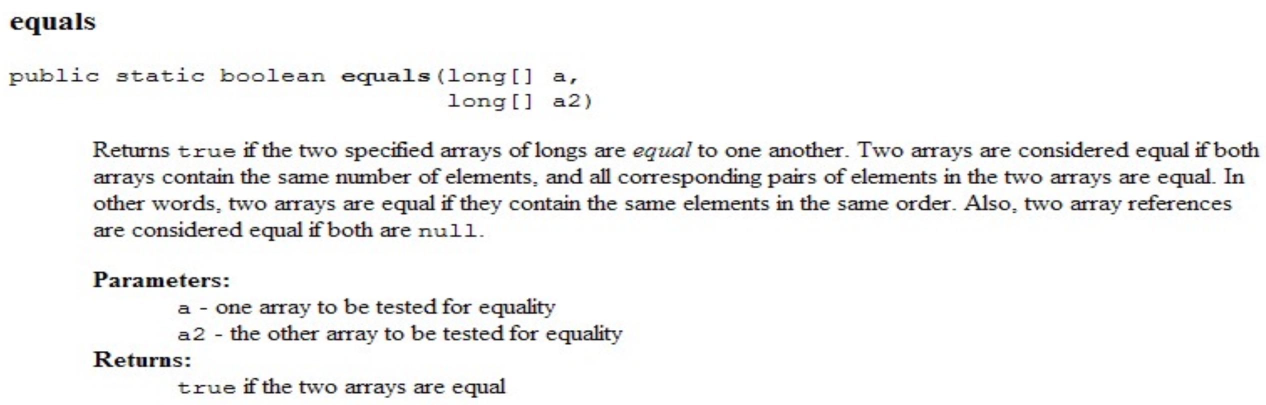
```
char[] array1 = {'A', 'B', 'C'};
char[] array2 = array1;
if (array1 == array2)
// do something
```

The condition in the if statement will evaluate to true, since both array variables refer to the same object.

```
char[] array1 = {'A', 'B', 'C'};
char[] array2 = {'A', 'B', 'C'};
if (array1 == array2)   // do something
```

Here the condition in the if statement will evaluate to false, since the array variables refer to **different** objects. It doesn't matter that the elements in the two objects have the same value; the== operator doesn't test this.

So, how do we test to see if two arrays are the same? We use a static method of the Array class called equals. To work with it, we need to import java.util.Arrays; at the top of our code. The API for the equals method shows:



equals

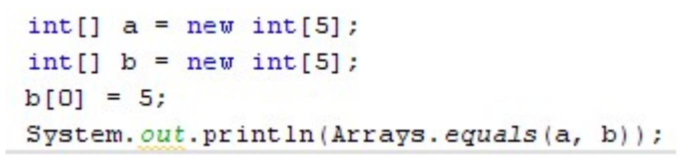
```
public static boolean equals(long[] a,
                             long[] a2)
```

Returns `true` if the two specified arrays of longs are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:
`a` - one array to be tested for equality
`a2` - the other array to be tested for equality

Returns:
`true` if the two arrays are equal

To use the method, we pass the names of two existing array variables. The method compares the array contents and returns true if they are equal and false if they are not. For example, the following code will print false since the arrays are not equal (since they are both integer arrays, all elements of both arrays will be initialized to zeroes, but we then change one element to a 5, which makes the arrays unequal):



```java
int[] a = new int[5];
int[] b = new int[5];
b[0] = 5;
System.out.println(Arrays.equals(a, b));
```

Recursion

When an algorithm has one subtask that is a smaller version of the entire algorithm's task, it is said to be recursive. A recursive algorithm could be used to write a recursive method. A recursive method contains an invocation of itself.

I have created a static method in MyUtility class to start with a certain number and print all the numbers from that number down to 0.

We will start by coding the base case for our countdown algorithm. A base case is a condition that stops the recursion and returns the result. If the number is less than or equal to zero, then print the number (which is always 0).

```java
public static void countDown(int number){
     if(number <= 0){
            System.out.println(number);
     }
}
```

To print the countdown, we have to invoke countDown(number - 1) recursively until it reaches down to 0.

```java
public static void countDown(int number){
     if(number <= 0){
            System.out.println(number);
     }else{
            System.out.println(number + ", ");
            countDown(number – 1);    ←recursive call
}
```

Let's say number is 2. We invoke countDown(2). Since 2 is not less than 0, control enters to else and prints 2 then call countDown(1). countDown(1) repeats the same process by printing 1 and call countDown(0). When countDown(0) is invoked, this executes the stopping/base case and ends the recursive method calls.