Arrays in Java

We looked at the basics of Java's built-in arrays, starting from creating an array to iterating over the elements of an array, as well as at a few typical array manipulations in class today. See the link to a complete implementation for you to play with at the end of this document.

Topics, in no particular order:

Creating an array

In Java, you create a new array in the following:

```
Foo[] x = new Foo[100];
```

Now `x` refers to an array of 100 `Foo` references. Note that the array doesn't actually contain the instances of `Foo` (the objects that is), but rather hold the references to the objects. We can assign other references to this array:

```
Foo[] y = x;
```

Now `y` is just another reference to the same array, and any change made through `y` will change the array referred to by `x`.

A Java array has a single attribute called `length` that gives you the capacity of the array; `x.length` for example would produce the value 100 in this case. The capacity of an array is fixed, and once created, cannot be changed. We "resize" and array by creating a new one with higher capacity, and copying the elements to the new one. See resizing an array for details.

Iterating over the elements of an array

Iterating over the elements of an array is the same as the following: `for each element v in the array x, do` *`something`* `with v`. There are two traditional *patterns* for iterating over the elements of an array: using a `while` or a `for` loop. Let's print the elements of the array `x` using a `while` loop (here the *`doing something`* is printing the element):

```
int i = 0;
while (i < x.length) {
    System.out.println(x[i]);
    i++;
}
```

And then using `for` loop:

```
for (int i = 0; i < x.length; i++)
    System.out.println(x[i]);
```

Java has another form of the `for` loop to implement this "foreach" pattern:

```
for (Foo v : x)
    System.out.println(v);
```

This version of the `for` has one advantage: it does exactly what it says - for each element v in the array x, it prints the element v! No indexing needed at all. It also has a disadvantage: there is no way to iterate over a part of the array, which is important when the *size* is not equal to the *capacity* of the array.

Copying an array

Copying the elements of a *source* array to *destination* array is simply a matter of copying the array element by element using an iterator.

```
public static Object[] copyArray(Object[] source) {
    Object[] copy = new Object[source.length];
    for (int i = 0; i < source.length; i++)
        copy[i] = source[i];

    return copy;
}
```

For you information, the `java.util.Arrays` class provides a set of methods to do just this for you in a very efficient way. Here's how:

```
public static Object[] copyArray(Object[] source) {
    Object[] copy = java.util.Arrays.copyOf(source, source.length);
    return copy;
}
```

Resizing an array

There is the classic problem of arrays - once created, it cannot change its capacity! The only way to "resize" an array is to first create a new and larger array, and copy the existing elements to the new array. The following static method resizes `oldArray` to have a capacity of `newCapacity`, and returns a

reference to the resized array.

```
static Object[] resize(Object[] oldArray, int newCapacity) {
    Object[] newArray = new Object[newCapacity];
    for (int i = 0; i < oldArray.length; i++)
        newArray[i] = oldArray[i];
    return newArray;
}
```

We can use this method in the following way:

```
Object[] data = new Object[10];
for (int i = 0; i < 10; i++)
    data[i] = new String(String.valueOf(i));

// The array "data" is now full, so need to resize before we can
// add more elements to it.
data = resize(data, 20);
for (int i = 10; i < 20; i++)
    data[i] = new String(String.valueOf(i));
```

Reversing an array

The simplest way of reversing an array is to first copy the elements to another array in the reverse order, and then copy the elements back to the original array. This *out-of-place* method is rather inefficient, but it's simple and it works.

```
public static void reverse(Object[] array) {
    Object[] tmpArray = new Object[array.length];
    int i = 0;                          // index into array
    int j = tmpArray.length - 1;        // index into the reverse copy
    while (i < array.length) {
        tmpArray[j] = array[i];
        i++;
        j--;
    }
    // Now copy the elements in tmpArray back into the original array.
    for (int i = 0; i < array.length; i++)
        array[i] = tmpArray[i];

    // NOTE: the following DOES NOT work! Why?
    // array = tmparray;
}
```

Fortunately, there is an *in-place* method that is far more efficient!

```
public static void reverse(Object[] array) {
    int i = 0;                          // forward index into left half
    int j = array.length - 1;           // backward index into right half
    while (i < j) {
        // Exchange array[i] with array[j]
        Object tmp = array[i];
```

```
            array[i] = array[j];
            array[j] = tmp;

            i++;
            j--;
        }
    }
```

Shifing an array left

Shifting an entire array left moves each element one (or more, depending how the shift amount) position to the left. Obviously, the first element in the array will *fall off* the beginning and be lost forever. The last slot of the array before the shift (ie., the slot where where the last element was until the shift) is now unused (we can put a `null` there to signify that). The size of the array remains the same however, because the assumption is that you would something in the now-unused slot. For example, shifing the array `[5, 3, 9, 13, 2]` left by one position will result in the array `[3, 9, 13, 2, -]`. Note how the `array[0]` element with value of `5` is now lost, and there is an empty slot at the end (shown as - above).

```
public static void shiftLeft(Object array[]) {
    for (int i = 1; i < array.length; i++)
        array[i - 1] = array[i];
    array[a.length - 1] = null;         // Now empty
}
```

What would happen if this were a *circular* or *cyclic* array? See show to rotate an array left.

Shifing an array right

Shifting an entire array right moves each element one (or more, depending how the shift amount) position to the right. Obviously, the last element in the array will *fall off* the end and be lost forever. The first slot of the array before the shift (ie., the slot where where the 1st element was until the shift) is now unused (we can put a `null` there to signify that). The size of the array remains the same however, because the assumption is that you would something in the now-unused slot. For example, shifing the array `[5, 3, 9, 13, 2]` right by one position will result in the array `[-, 5, 3, 9, 13]`. Note how the `array[4]` element with value of `2` is now lost, and there is an empty slot in the beginning (shown as - above).

```
public static void shiftRight(Object array[]) {
    for (int i = array.length - 1; i > 0; i--)
        array[i] = array[i - 1];
    array[0] = null;                    // Now empty.
}
```

What would happen if this were a *circular* or *cyclic* array? See show to rotate an array right.

Inserting an element into an array

Inserting an element into any slot in an array requires that we first make *room* for it by shifting some of the elements to the right, and then insert the new element in the newly formed *gap*. The only time we

don't have to shift is when we insert in the next available empty slot in the array (the one after the last element). The insertion also assumes that there is at least one empty slot in the array, or else it must be resized as we had done earlier (or, if it's *non-resizable* by policy, then we can throw an exception). For example, inserting the value `7` in the slot with index `2` in the array `[3, 9, 12, 5]` produces the array `[3, 9, 7, 12, 5]`.

```
// Insert the given element at the given index in the non-resizable array
// with size elements.
public static void insert(Object[] array, int size, Object elem, int index) {
    if (size == array.length)
        throw new RuntimeException("no space left");
    else {
        // make a hole by shifting elements to the right.
        for (int i = index; i < size; i++)
            array[i + 1] = array[i];

        // now fill the hole/gap with the new element.
        array[index] = elem;
    }
}
```

Removing an element from an array

After removing the element at a given index, we need to *plug the hole* by shifting all the elements to its right one position to the left.

```
// Removes the element at the given index from the array with size elements.
public static void remove(Object[] array, int size, int index) {
    // Shift all elements [index+1 ... size-1] one position to the
    // left.
    for (int i = index + 1; i < size; i++)
        array[i - 1] = array[i];

    // Now nullify the unused slot at the end.
    array[size - 1] = null;
}
```

Rotating an array left

Rotating an array left is equivalent to shifting a *circular* or *cyclic* array left — the 1st element will not be lost, but rather move to the last slot. Rotating the array `[5, 3, 9, 13, 2]` left by one position will result in the array `[3, 9, 13, 2, 5]`.

```
public static void rotateLeft(Object array[]) {
    Object firstElement = array[0];
    for (int i = 1; i < array.length; i++)
        array[i - 1] = array[i];
    array[a.length - 1] = firstElement;
}
```

There is a much more elegant solution that we'll see when we discuss *circular* or *cyclic* arrays (wait till we study the Queue ADT).

Rotating an array right

Rotating an array right is equivalent to shifting a *circular* or *cyclic* array right — the last element will not be lost, but rather move to the 1st slot. Rotating the array `[5, 3, 9, 13, 2]` right by one position will result in the array `[2, 5, 3, 9, 13]`.

```
public static void rotateRight(Object array[]) {
    Object lastElement = array[array.length - 1];
    for (int i = array.length - 1; i > 0; i--)
        array[i] = array[i - 1];
    array[0] = lastElement;
}
```

There is a much more elegant solution that we'll see when we discuss *circular* or *cyclic* arrays (wait till we study the Queue ADT).

You can look at the example ArrayExamples.java class to see how these can be implemented. Run the `ArrayExamples.main()` to see the output.