Welcome to the Thing Array project. This is a comprehensive exercise in creating and maintaining an array of objects that can grow, shrink, and sort itself as needed. Cool beans!

This is "Pair Programming" Assignment

While you are working on this project, you will use the "Pair Programming" approach. In pair programming, two programmers share one computer. One student is the "driver," who controls the keyboard and mouse. The other is the "navigator," who observes, asks questions, suggests solutions, and thinks about slightly longer-term strategies. I would suggest that the 'navigator' have the directions open on their screen, and the 'driver' have the code on their screen. The two programmers should switch roles about every 20 minutes. Working in pairs should make you much better at programming than would be by working alone. The resulting work of pair programming nearly always outshines that of the solitary programmer, with pairs producing better code in less time. For this assignment each partner will receive the same grade.

When you have selected your partner and you are sitting next to each other, you may begin work on this assignment. Remember that you both need to be working together as a unified pair in order to successfully complete this assignment. No 'going rogue' on your own. Good luck and have fun!

Be sure that both of you have duplicate saved copies of your work each day. You need to have something to work with tomorrow in the event your partner is absent.

A `ThingArray` is an array of `Thing` objects. `Thing` objects contain primitive `int` values, and implement the `Comparable` interface, so that they can be compared to each other and sorted in order. You will first be completing the `Thing` class, and then writing the `ThingArray` class from scratch according to the instructions below. You will know when you have completed the assignment when you can successfully run `ThingArrayTest`.

You can get started on this assignment by downloading, extracting, and saving the Thing Array project to your computer.

First complete the `Thing` class. `Thing` objects are pretty basic; they have one instance variable `amount` that keeps track of an `int` value.

- **Write the constructor.** You can start out by writing a `Thing` constructor that requires an integer parameter, and then sets the instance variable.

- **Write the `getAmount()` method.** This method returns an integer. Simply return the instance variable `amount`.

- **Complete the `compareTo` method.** `Thing` objects also implement the `Comparable` interface, which requires them to override the `compareTo` method from the `Object` class. In order to do so, the overridden method header in this class must be exactly the same as the one from the `Object` class, which passes an `Object` as a parameter. We don't want to compare `Objects`, we want to compare `Things`, so the first thing the method does is downcast the `Object` object back into a `Thing` object. `Thing` objects are compared to other `Thing` objects based on their `amount`. `thing1.compareTo(thing2)` returns a positive integer if `thing1` is greater than `thing2` (i.e `thing1.getAmount() - thing2.getAmount()`), and vice versa. If both `Thing` objects are equivalent, then `thing1.compareTo(thing2)` will be zero. The `compareTo` method is partially completed for you so that you can easily finish it. All you need to do is `return this.getAmount() - other.getAmount()`.

- **Write the** `equals(Thing other)` **method.** `Thing` objects also are required to have an `equals` method that compares two `Thing` objects to see if they are equivalent (i.e. two separate `Thing` objects with the same `amount` values). The `equals` method receives a `Thing` object as a parameter. Since you just wrote `compareTo`, this one is easy. The `equals` method should return `true` if `compareTo` is zero.

Once you have completed these four items, run `ThingArrayTest` to verify that your `Thing` class is complete (the tester class won't be fully satisfied, but you will at least be able to check that `Thing` is correct).

Next, begin to write the `ThingArray` class from scratch (e.g. create a new `ThingArray` class). Below is an outline of what needs to be done to complete your `ThingArray` class. Note that for this first part of the assignment, you won't have to worry about the array filling up to capacity (we'll worry about that later on).

- **Declare the instance variables.** `ThingArray` objects have the following instance variables:
    - `String name` (the name of the array)
    - `Thing[] list` (the array of Thing objects)
    - `int size` (the logical size of the array)

  Of course, all of these instance variables should be `private`.

- **Write the** `ThingArray` **constructor**. The constructor takes a `String` parameter (the name of the array), and should set the instance variables. `list` *must* be instantiated to a length of 4, and `size`, of course will be zero (because no `Thing` objects have yet been stored in `list`).

- **Write the** `getName()` **method**. The method returns a `String` object. Simply return the instance variable `name`.

- **Write the** `getSize()` **method**. The method returns an integer. Simply return the instance variable `size`.

- **Write the** `isEmpty()` **method**. The method returns true or false. Check if `size` is equal to zero.

- **Write the** `isFull()` **method**. The method returns true or false. Check if `size` is equal to the length of the array.

- **Write the** `add(Thing element)` **method**. The method does not return anything. This method appends the specified element to the end of the list. The instance variable `size` represents the index of the next available spot in the array. Don't forget to increment `size` when you are done.

- **Write the** `get(index)` **method**. The method returns a `Thing` object. Return the element of the array at the given index.

- **Write the** `indexOf(Thing element)` **method**. The method returns an integer. Iterate though the logical `size` of the array (just the data, not the empty spots [`null` pointers] at end), searching for the given `Thing` object. (Don't forget about the appropriate way to compare the equivalence of two objects.) Return either the index of the first occurrence of the object, or else -1 if it is not found.

- **Write the** `contains(Thing element)` **method**. The method returns true if the list contains the specified element, otherwise false. Since you just wrote `indexOf`, this one is easy. Just check if `indexOf` is -1 or not.

- **Write the** `toString()` **method**. The method returns (of course) a `String` object. This method is not implicitly checked by the `ThingArrayTest`, but you will want to write something useful here for your own later debugging purposes. Simply iterate through the array, creating a `String` that displays the index and the `getAmount` value of each `Thing` object in the array (put each entry on a new line). Remember that a `toString` method returns a `String` object, and does *not* print anything to the console window.

- **Write the** `set(int index, Thing element)` **method**. The method returns a `Thing` object. This method replaces the `Thing` object at the given index. Notice that it returns the `Thing` object that was there before the replacement, so you may want to save it to a temporary local variable that you can then return when you are finished with the replacement.

Now the methods get a little bit more challenging to write. The time has come to deal with an array that is filled to capacity. The general rule is that for an array that is full, we want to create a new array that has twice the capacity, copy over each element (one at a time) to the new array, and then point (reassign the variable) from the old array to the new array.

On a similar note, when an array is more than three-quarters empty, there is a lot of wasted (unused) space. The general rule here is to create a new array that is one-half the capacity of the original array, copy over each element (one at a time) to the new array, and then point (reassign the variable) from the old array to the new array.

There are several methods in the `ThingArray` class that could cause us to go over or under capacity (namely the various versions of `add` and r`emove`). It would be better to write a private helper method that all of these methods could call before they add or after they remove items to see if the array should be resized. Here is a combination of code and pseudocode (I'm not going to give it all away) that outlines the process.

```
private void checkSize()
{
  if array is full
    newSize = list.length * 2;  // double the array capacity
  else if (size < list.length / 4)
    newSize = list.length / 2;  // halve the size of the array
  else
    return;                     // leave the array alone
```

```
        Thing[] temp = new Thing[newSize]
        loop through the old array
           copy over each of the elements to new array
        list = temp;
     }
```
Once you have written the method, then call it *before you add anything*, and *after you remove anything*.

Remember that to insert an element into the middle of an array, you have to 'part the waters' (make an opening), starting by moving all the elements to the right up one position, and that when you remove an element from the middle of the array, you have to 'close the curtains' (eliminate the space), by moving all the elements on the right down one position. Be sure to do the movements in the right order.

- **Write the** `add(int index, Thing element)` **method**. This method does not return anything. This inserts a `Thing` object at the given index, after first moving everything down one position. Remember to work from the end of the array data (the logical size not necessarily the end of the array), and work backwards to the given index position. Don't forget to increment `size` when you are done.

- **Write the** `remove(int index)` **method**. This method returns a `Thing` object. It removes the `Thing` object at the given index. Notice that it returns the `Thing` object that was there before the removal, so you may want to save it to a temporary local variable that you can then return when you are finished with the removal. The remove operation is rather simple; just move everything to the right of the index down one position, which eliminates the given item. Remember to set the last element to `null` so there won't be a duplicated reference at the end. Don't forget to decrement `size`, and then check if the array capacity should be made smaller.

- **Write the** `remove(Thing element)` **method**. This method returns true or false. This method removes the first occurrence of the specified element, if it exists. If the specified element is found and removed, the method returns true, otherwise false (it was not found). Since you have already written the `remove(index)` method, you just need to find the `indexOf` (hint, hint) the `Thing` you are looking for, and then remove the `Thing` at that index (provided it is not -1).

- **Write the** `clear()` **method.** This method does not return anything. As the name implies, the clear method removes all of the Thing objects from the array. One way is to create a brand new, empty array, but perhaps an even easier way is just to keep removing the first element in the array as long as the array `size` is greater than zero.

- **Write the** `sort()` **method**. This method does not return anything. Use any algorithm of your choice (feel free to look in the book for code examples) to sort the array of `Thing` objects in ascending order. Note that `ThingArrayTest` will generate 100 random `Thing` objects and then test to see if they are in sorted order after calling this method. If your sort method does not work properly, `ThingArrayTest` will display the test array elements (using your `toString` method) so that you can see how you are doing.

Congratulations on completing this challenging assignment. When you are finished, please demonstrate your work to me by successfully running the provided `ThingArrayTest` class.