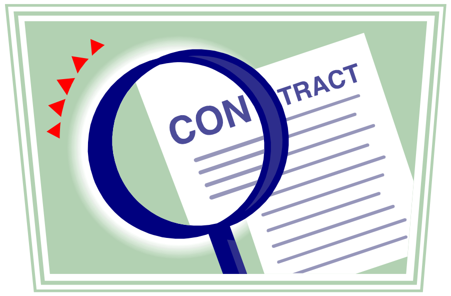
Chapter 1111: Implementing a Collection Class

WEEK 3: CREATING THE ArrayIntList CLASS

NOTE:

The full implementations of ArrayIntList and ArrayList<E> are not covered in these slides. Please refer to your textbook and to the files posted in Canvas for the full class implementations.



ArrayIntList Class

- In this chapter we will discuss two implementations of the ArrayList class:
  - **ArrayIntList** – simpler version that only stores ints
  - **ArrayList<E>** – generic version
- When implementing a new class, two perspectives need to be considered:
  - The client's view – wants to know **WHAT** the class does (clear specifications), **NOT** generally interested in implementation details
  - The implementer's view – needs to know how to make the object work and how to ***satisfy the contract*** with the client

Class fields

- Before doing anything, you need to think about the fields you want to include in your class
- Field 1: Since we are implementing ArrayIntList, we need an integer array:

```
private int[] elementData;
```

- The array can be initialized using a constant, e.g.:

```
public static final int DEFAULT_CAPACITY = 100;
```

- Field 2: An int variable size will keep track of the number of items stored in the array:

```
private int size;
```

Add a constructor (or two…)

- No-parameter constructor using the default array size:

```
public ArrayIntList(){
        elementData = new int[DEFAULT_CAPACITY];
        size = 0;
}
```

- This constructor will work, but it isn’t very flexible – what if the client had more than 100 elements to store?
- A **second**, more **general**, constructor can be written to take a parameter ***specifying the capacity*** of the array

Adding a 2nd constructor

- The second constructor accepts an integer as its input parameter and uses this value to set the capacity of the array:

```
public ArrayIntList(int capacity){
        elementData = new int[capacity];
        size = 0;
}
```

- This gives the client the flexibility to create a list of any capacity by writing code such as:

```
ArrayIntList myList = new ArrayIntList(250);
```

Avoiding Redundancy

- Want to avoid introducing redundant code
- A good way to do this is to have the less general constructor call the more general one
- In this case, the no-parameter constructor can be rewritten to call the second, more flexible constructor:

```
public ArrayIntList(){
        this(DEFAULT_CAPACITY);
}
```

How does this rewritten constructor work?

- Java is able to distinguish between the two constructors because they have ***different signatures***
- When Java sees the call "this(DEFAULT_CAPACITY)" in the no-parameter constructor, it knows to call the 2nd constructor (since that is the one that includes an integer value as a parameter)
- "this" refers to this instance of an ArrayIntList object that is being constructed

Adding elements to an ArrayIntList object

- Now that we have constructors, we can start adding some class methods
- A good method to start with is a method to add integers to the list
- We probably want two add methods:
  - One that adds the element to the end of the list
  - One that adds the element at a specified index

Adding items to the list

```
public void add(int value){
        elementData[size] = value;
         size++;
}
```

- The new value gets added at index = size (i.e. the end of the list)

```
public void add(int index, int value){
     for(int i = size; i >= index + 1; i--){
          elementData[i] = elementData[i-1];
     }
      elementData[index] = value;
      size++;
}
```

- The loop runs backwards to avoid overwriting list values
- The new value is added at the specified index

Printing an ArrayIntList object

- Now that we have methods to construct lists and to add items to them, it might be nice to be able to print them
- Since the array generally won't be completely full, we will use the size field's value in our for loop:

```
public void toString(){
      if(size == 0){
            return “[]“;
      } else {
             String result = “[“ + elementData[0];
             for(int i = 1; i < size; i++){
                     result += “, “ + elementData[i];
              }
              result += “]”;
              return result;
       }
   }
```

Preconditions & Postconditions

- The new class and all its methods should be ***well documented*** so that anyone using the class (either as a client or as an implementer) knows what each method does and what limitations it has
- Preconditions are assumptions that the method makes. They give you the opportunity to describe any **dependencies** that the method has.
- Postconditions describe what the method accomplishes, ***assuming the preconditions are met***.

Example using the get method

```
//pre: 0 <= index < size()
//post: returns the integer at the given index in the list
public int get(int index){
    return elementData[index];
}
```

- The precondition ensures the index specified is legal
- Method comments such as preconditions and postconditions should not contain implementation details – those should be reserved for comments within the method code itself

Throwing exceptions

- While documenting preconditions is good practice, we can not assume that clients will always satisfy these preconditions
- The convention in such cases is to throw an exception
- For example, if a constructor is called with a negative integer as its input parameter, it would be appropriate to throw an IllegalArgumentException:

```
if(capacity < 0){
    throw new IllegalArgumentException():
}
```

- The exception can be made more informative by including a string within the parentheses that will be displayed with the exception
- The exact type of exception thrown should be mentioned in the comments for the method, along with the circumstances that would cause it

Constructors (final version)

```
//post: constructs an empty list of default capacity
Public ArrayIntList(){
    this(DEFAULT_CAPACITY);
}
```

```
//pre: capacity >= 0 (throws IllegalArgumentException if not)
//post: constructs an empty list with the given capacity
public ArrayIntList(int capacity){
    if(capacity < 0){
        throw new IllegalArgumentException(“capacity: “ + capacity);
    }
    elementData = new int[capacity];
    size = 0;
}
```

Common Exception Types

Exception Type	Description
NullPointerException*	A null value has been used in a case that requires an object
ArrayIndexOutOfBoundsException*	A value passed as an index to an array is illegal
IndexOutOfBoundsException	A value passed as an index to some nonarray structure is illegal
IllegalStateException	A method has been called at an illegal or inappropriate time
IllegalArgumentException	A value passed as an argument to a method is illegal
NoSuchElementException	A call was made on an iterator's next method when there were no values left to iterate over

* These exceptions are thrown automatically by Java & don't generally need to be specified

Checking conditions using private methods

- private methods (methods used only within the class, not by the client) can be used to ensure that other methods won't fail
- shown below are two such private methods that would be useful to include in our ArrayIntList class:

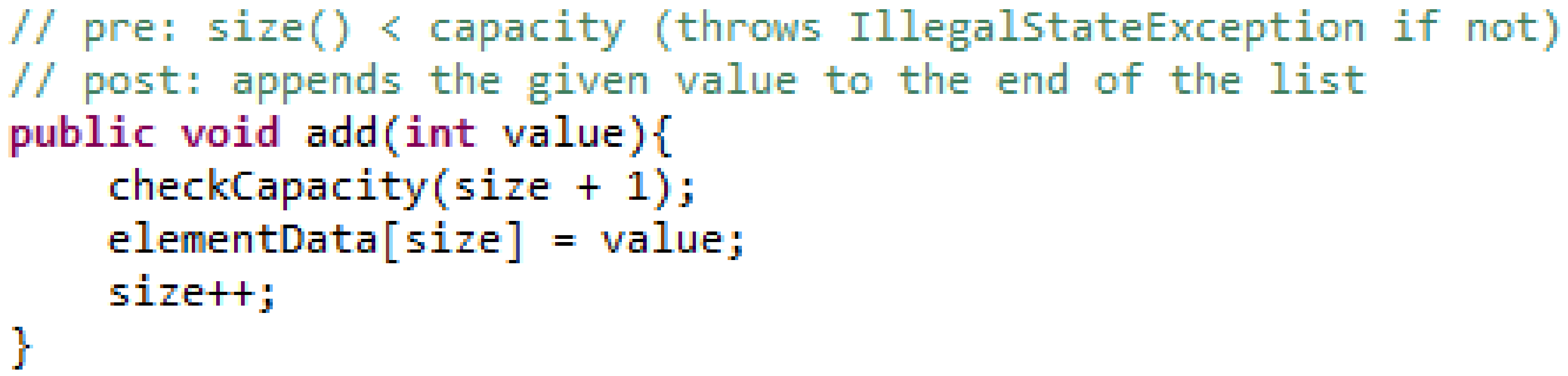
Can use with the add methods:

```
// post: checks that the underlying array has the given capacity,
//       throwing an IllegalStateException if it does not
private void checkCapacity(int capacity){
    if (capacity > elementData.length){
        throw new IllegalStateException("exceeds list capacity");
    }
}
```

Can use with the get and remove methods:

```
// post: throws an IndexOutOfBoundsException if the given index is
//        not a legal index of the current list
private void checkIndex(int index){
    if (index < 0 || index >= size){
        throw new IndexOutOfBoundsExcption("index: " + index);
    }
}
```

Revised add method



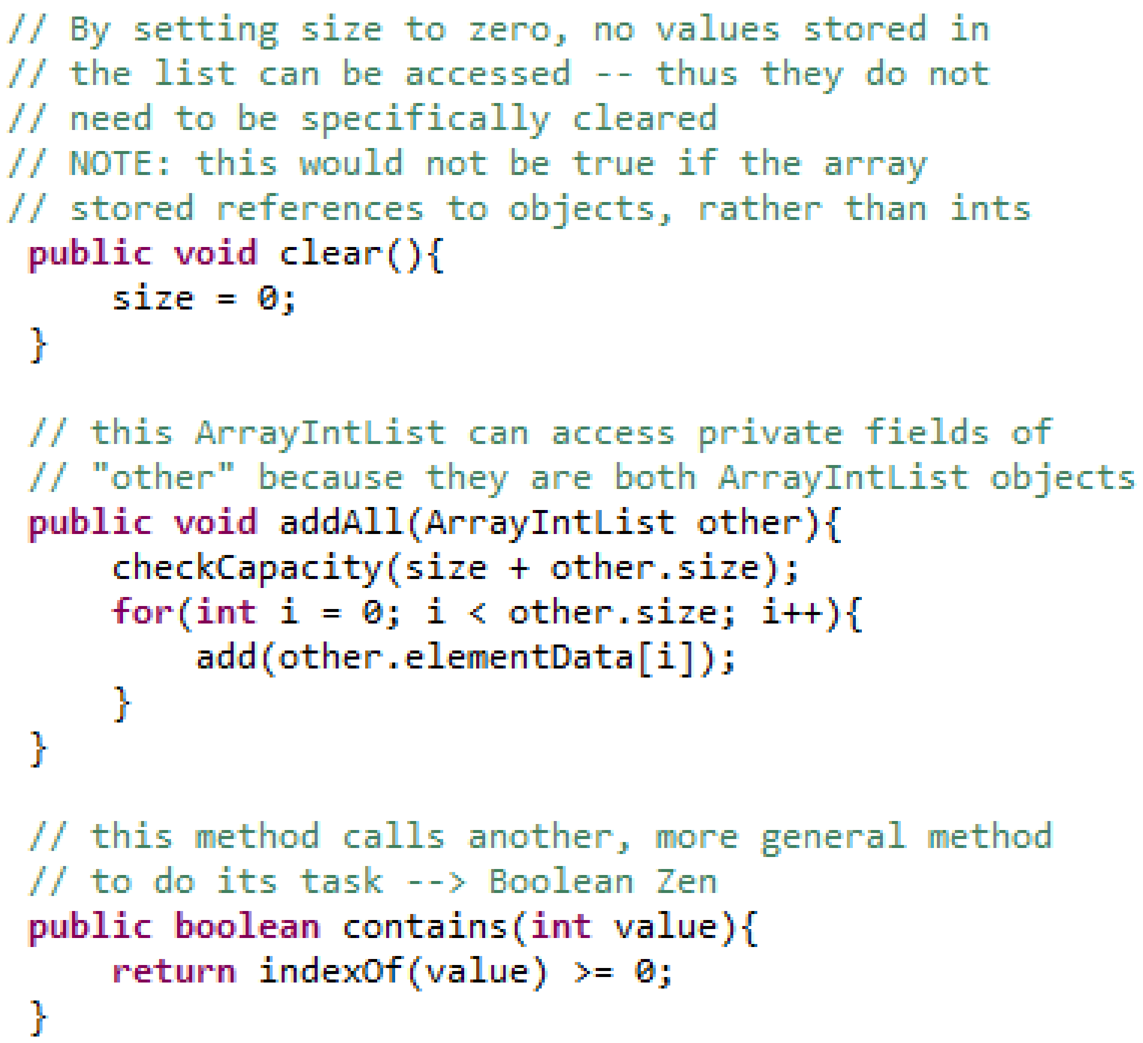
```
// pre: size() < capacity (throws IllegalStateException if not)
// post: appends the given value to the end of the list
public void add(int value){
    checkCapacity(size + 1);
    elementData[size] = value;
    size++;
}
```

Other points of interest

Class	Take note of:
clear()	Since our arrays contain integers, rather than references to objects, nothing needs to be set to null (no objects available for garbage collection). All that needs to be done to “clear” the list is to set the size to 0, cutting off access to any elements until new elements are added.
addAll()	This method can access the private fields of another ArrayIntList object without using a set method because “private” means “private to the class”. Since both ArrrayIntList objects “this” and “other” are objects of the same class, they can access each other’s private fields.
contains()	This method uses Boolean Zen and calls another, more specific class method to perform the requested task.

These methods will be shown on the next slide…

Interesting methods



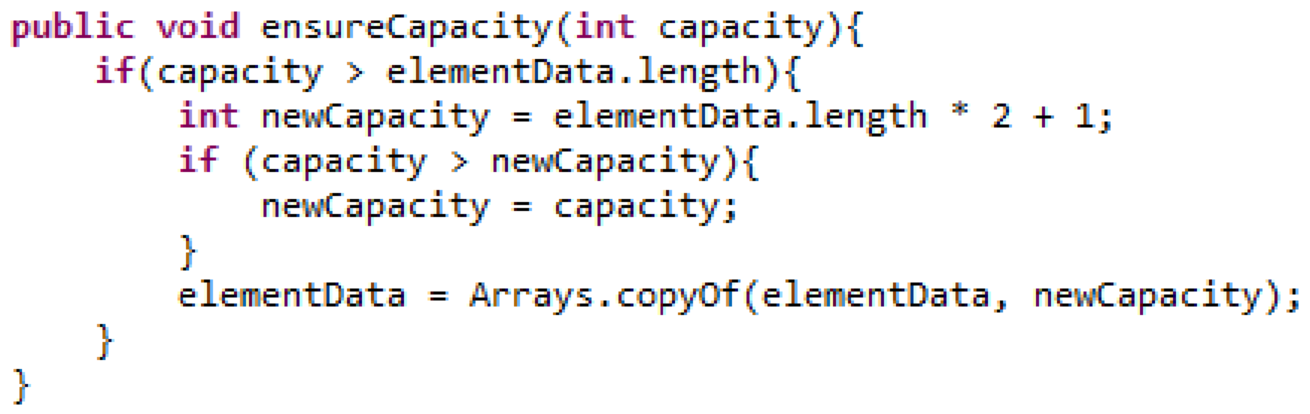
```java
// By setting size to zero, no values stored in
// the list can be accessed -- thus they do not
// need to be specifically cleared
// NOTE: this would not be true if the array
// stored references to objects, rather than ints
 public void clear(){
     size = 0;
 }

 // this ArrayIntList can access private fields of
 // "other" because they are both ArrayIntList objects
 public void addAll(ArrayIntList other){
     checkCapacity(size + other.size);
     for(int i = 0; i < other.size; i++){
         add(other.elementData[i]);
     }
 }

 // this method calls another, more general method
 // to do its task --> Boolean Zen
 public boolean contains(int value){
     return indexOf(value) >= 0;
 }
```

Resizing

- You probably remember that the ArrayList grew in capacity as needed. Our ArrayIntList should do the same.
- We can use a built-in method called Arrays.copyOf to help us accomplish this – this method returns the values of the original array copied into a new (larger) array.



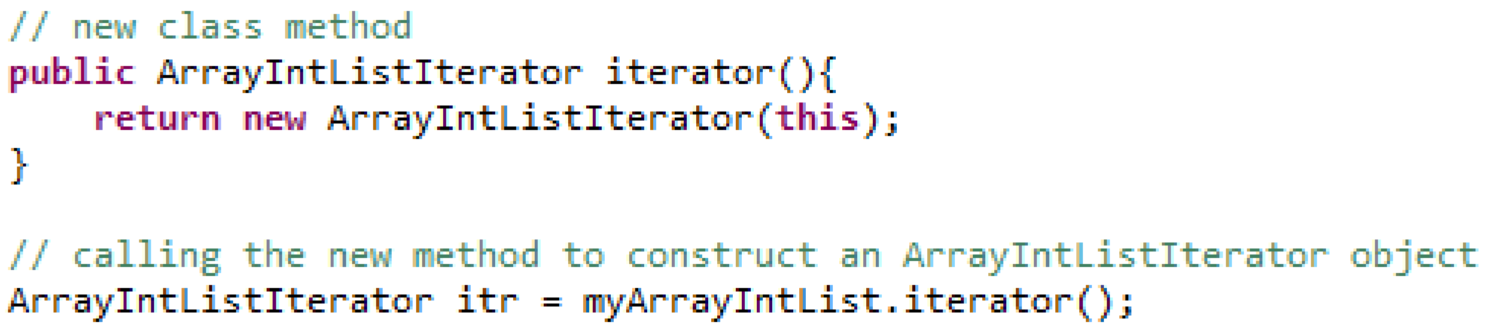
```
public void ensureCapacity(int capacity){
    if(capacity > elementData.length){
        int newCapacity = elementData.length * 2 + 1;
        if (capacity > newCapacity){
            newCapacity = capacity;
        }
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

Adding an iterator

- It is common for collections in the Java Collections Framework to use an iterator object to traverse a collection
- An iterator should provide three basic operations, all of which you have seen before: hasNext(), next(), and remove()
- The code to implement an ArrayIntListIterator class is provided on pages 938-939 in the text and as files (.java and .txt) on the website.

The iterator class

- In addition to constructing a new class, the ArrayIntListIterator class, a new method will need to be added to the ArrayIntList class itself – a method to construct the iterator:



```
// new class method
public ArrayIntListIterator iterator(){
    return new ArrayIntListIterator(this);
}

// calling the new method to construct an ArrayIntListIterator object
ArrayIntListIterator itr = myArrayIntList.iterator();
```

- By using the “this” keyword, we are asking the method to construct an iterator that will iterate over the list on which it was called

Encapsulation

- When we declared the fields of our ArrayIntList class, we followed the principles of encapsulation and declared them to be private.
- Because these fields are private, the client cannot access them directly.
- Accessor methods give the client access to the data stored in these private fields while allowing the class to control this access:
  - Get methods do not change field values
  - Set methods can include checks before making any changes to field values

ArrayList<E>

- To convert ArrayIntList to ArrayList<E>, it will be sufficient in most instances to replace occurrences of ArrayIntList with ArrayList<E>
- HOWEVER… there are a few places where this won't work
- For example, it is **not possible** to construct generic arrays
- Constructing an array of type Object **IS** possible
- Thus, we can create an array of type Object and use a cast:

```
elementData = (E[]) new Object[capacity];
```

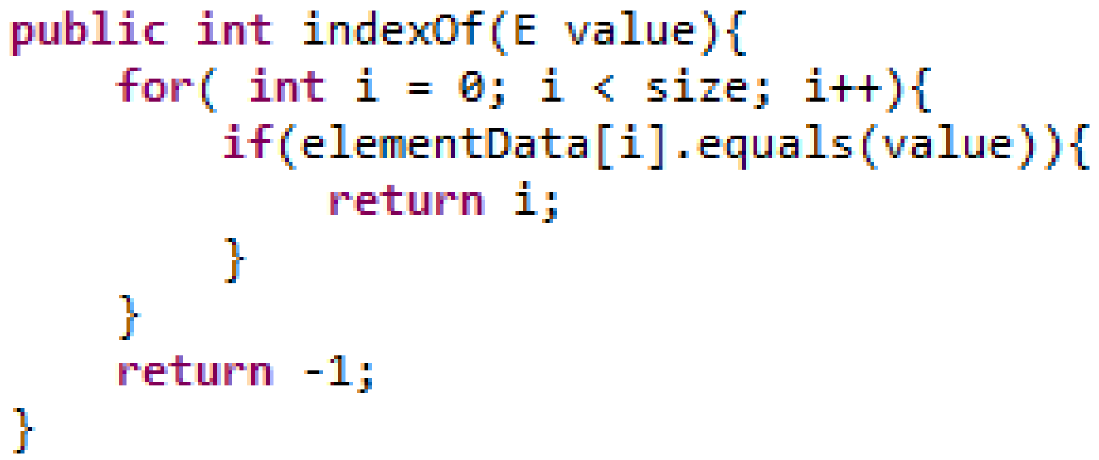
ArrayList<E>, continued…

- Potential problems also exist with the **indexOf** method
- When we had an ArrayIntList, comparing values meant comparing ints, so we could use "**==**"
- Now that we are comparing objects, this won't work
- Instead, we should use an **equals** method:

e.g.

```
if (elementData[i].equals(value)){ … }
```

Revised indexOf method



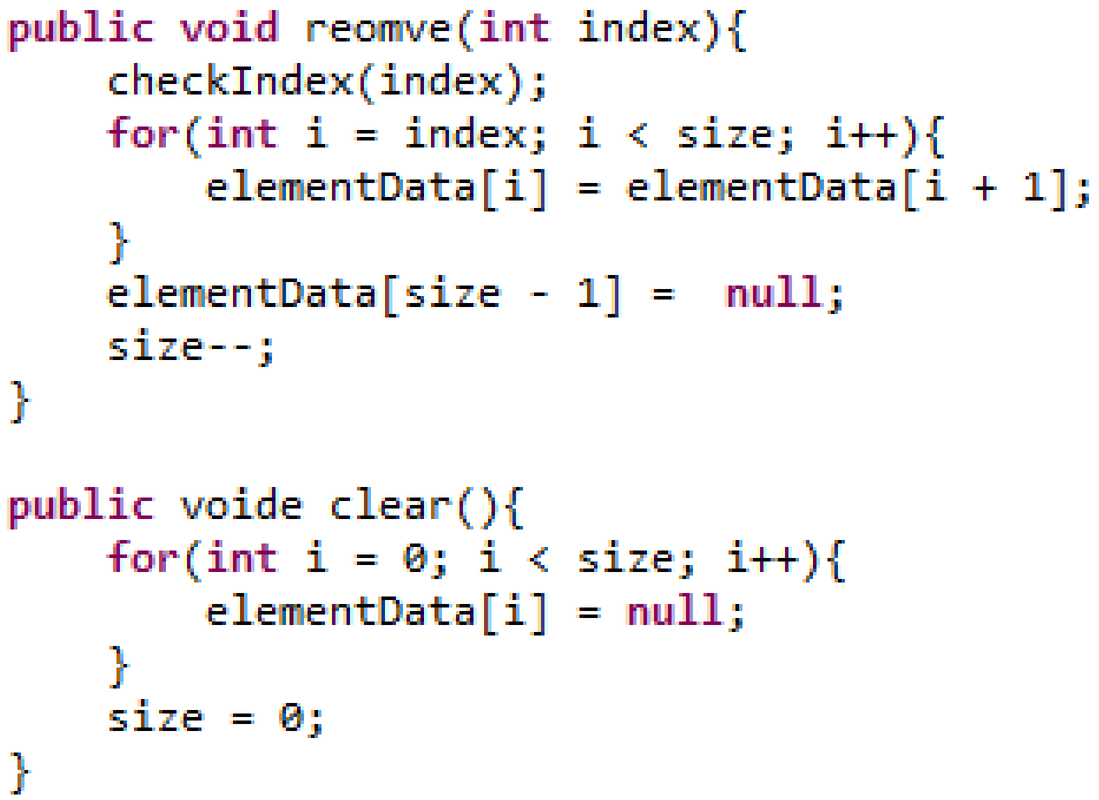
```
public int indexOf(E value){
    for( int i = 0; i < size; i++){
        if(elementData[i].equals(value)){
            return i;
        }
    }
    return -1;
}
```

Using “==“ in this method would require that the two objects being compared were the same unique object, rather than two objects with equivalent values

Memory allocation & Garbage collection

- When an object is no longer needed as an ArrayList element, it should be removed by the Java garbage collector.
- The garbage collector looks for objects that are no longer being used.
- If the ArrayList is still keeping references to these elements, the garbage collector won’t “recognize” them and the space they occupy won’t be reclaimed.
- To “free up” an object for garbage collection, values should be explicitly set back to **null**.

Modify remove and clear methods for garbage collection



```
public void reomve(int index){
    checkIndex(index);
    for(int i = index; i < size; i++){
        elementData[i] = elementData[i + 1];
    }
    elementData[size - 1] =  null;
    size--;
}

public voide clear(){
    for(int i = 0; i < size; i++){
        elementData[i] = null;
    }
    size = 0;
}
```

After the elements in the array have been shifted following the removal of an element, the formerly last occupied space in the array now holds an unused value that needs to be set to null.

To clear an array, all the formerly occupied spaces now contain values that should be reset to null.

Modifications to the iterator

- Please refer to the file linked on the website for implementation
- The iterator class is converted to an inner class (a class declared inside another class).
- This leads to some interesting points in how the inner class is declared:

```
Public class ArrayList<E>{
    . . .
    private class ArrayListIterator implements Iterator<E>{
        . . .
    }
}
```

- The inner class is declared without the "<E>" because the type is already declares as part of the outer class.
- However, we still need to say the class implements Iterator<E> to satisfy the interface requirements.