

CSE 1201

Object Oriented Programming

ArrayList

Acknowledgement

□ For preparing the slides I took materials from the following sources

- Course Slides of Dr. Tagrul Dayar, Bilkent University
- Java book “*Java Software Solutions*” by Lewis & Loftus.

What's an Array List

□ **ArrayList** is

- a class in the standard Java libraries that can hold any type of object
- an object that can grow and shrink while your program is running (unlike arrays, which have a fixed length once they have been created)

□ In general, an **ArrayList** serves the same purpose as an array, except that an **ArrayList** can change length while the program is running

The ArrayList Class

- The class **ArrayList** is implemented using an array as a private instance variable
 - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array

Using the `ArrayList` Class

- ❑ In order to make use of the `ArrayList` class, it must first be imported

```
import java.util.ArrayList;
```

- ❑ An `ArrayList` is created and named in the same way as object of any class, except that you specify the base type as follows:

```
ArrayList<BaseType> aList =  
    new ArrayList<BaseType>();
```

Creating an ArrayList

□ An initial capacity can be specified when creating an **ArrayList** as well

- The following code creates an **ArrayList** that stores objects of the base type **String** with an initial capacity of 20 items

```
ArrayList<String> list = new ArrayList<String>(20);
```

- Specifying an initial capacity does not limit the size to which an **ArrayList** can eventually grow

□ Note that the base type of an **ArrayList** is specified as a *type parameter*

Adding elements to an ArrayList

- The **add** method is used to add an element at the “end” of an **ArrayList**

```
list.add("something");
```

- The method name **add** is overloaded
- There is also a two argument version that allows an item to be added at any currently used index position or at the first unused position

How many elements?

- The **size** method is used to find out how many indices already have elements in the **ArrayList**

```
int howMany = list.size();
```

- The **set** method is used to replace any existing element, and the **get** method is used to access the value of any existing element

```
list.set(index, "something else");
```

```
String thing = list.get(index);
```

- **size** is NOT capacity
 - size is the number of elements currently stored in the **ArrayList**
 - Capacity is the maximum number of elements which can be stored. Capacity will automatically increase as needed

ArrayList code Example

```
// Note the use of Integer, rather than int
public static void main( String[ ] args)
{
    ArrayList<Integer> myInts = new ArrayList<Integer>(25);
    System.out.println( "Size of myInts = " + myInts.size());
        for (int k = 0; k < 10; k++)
            myInts.add( 3 * k );
            myInts.set( 6, 44 );
    System.out.println( "Size of myInts = " + myInts.size());
        for (int k = 0; k < myInts.size(); k++)
            System.out.print( myInts.get( k ) + ", " );
        }
    // output
        Size of myInts = 0
        Size of myInts = 10
    0, 3, 6, 9, 12, 15, 44, 21, 24, 27
```

Methods in the Class `ArrayList`

- The tools for manipulating arrays consist only of the square brackets and the instance variable `length`
- `ArrayLists`, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays

Some Methods in the Class `ArrayList` (Part 1 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

CONSTRUCTORS

```
public ArrayList<Base_Type>(int initialCapacity)
```

Creates an empty `ArrayList` with the specified *Base_Type* and initial capacity.

```
public ArrayList<Base_Type>()
```

Creates an empty `ArrayList` with the specified *Base_Type* and an initial capacity of 10.

(continued)

Some Methods in the Class `ArrayList` (Part 2 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

ARRAYLIKE METHODS

```
public Base_Type set( int index, Base_Type newElement)
```

Sets the element at the specified `index` to `newElement`. Returns the element previously at that position, but the method is often used as if it were a `void` method. If you draw an analogy between the `ArrayList` and an array `a`, this statement is analogous to setting `a[index]` to the value `newElement`. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws an `IndexOutOfBoundsException` if the `index` is not in this range.

```
public Base_Type get(int index)
```

Returns the element at the specified `index`. This statement is analogous to returning `a[index]` for an array `a`. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range.

(continued)

Some Methods in the Class `ArrayList` (Part 3 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

METHODS TO ADD ELEMENTS

```
public boolean add(Base_Type newElement)
```

Adds the specified element to the end of the calling `ArrayList` and increases the `ArrayList`'s size by one. The capacity of the `ArrayList` is increased if that is required. Returns `true` if the add was successful. (The return type is `boolean`, but the method is typically used as if it were a `void` method.)

```
public void add( int index, Base_Type newElement)
```

Inserts `newElement` as an element in the calling `ArrayList` at the specified index. Each element in the `ArrayList` with an index greater or equal to `index` is shifted upward to have an index that is one greater than the value it had previously. The index must be a value greater than or equal to 0 and less than *or equal* to the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the index is not in this range. Note that you can use this method to add an element after the last element. The capacity of the `ArrayList` is increased if that is required.

(continued)

Some Methods in the Class `ArrayList` (Part 4 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

METHODS TO REMOVE ELEMENTS

```
public Base_Type remove(int index)
```

Deletes and returns the element at the specified index. Each element in the `ArrayList` with an index greater than `index` is decreased to have an index that is one less than the value it had previously. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range. Often used as if it were a void method.

(continued)

Some Methods in the Class ArrayList (Part 5 of 11)

Display 14.1 Some Methods in the Class ArrayList

```
protected void removeRange(int fromIndex, int toIndex)
```

Deletes all the element with indices i such that $\text{fromIndex} \leq i < \text{toIndex}$. Element with indices greater than or equal to toIndex are decreased appropriately.

```
public boolean remove(Object theElement)
```

Removes one occurrence of theElement from the calling ArrayList. If theElement is found in the ArrayList, then each element in the ArrayList with an index greater than the removed element's index is decreased to have an index that is one less than the value it had previously. Returns true if theElement was found (and removed). Returns false if theElement was not found in the calling ArrayList.

```
public void clear()
```

Removes all elements from the calling ArrayList and sets the ArrayList's size to zero.

(continued)

Some Methods in the Class ArrayList (Part 6 of 11)

Display 14.1 Some Methods in the Class ArrayList

SEARCH METHODS

```
public boolean contains(Object target)
```

Returns true if the calling ArrayList contains target; otherwise, returns false. Uses the method equals of the object target to test for equality with any element in the calling ArrayList.

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to target. Uses the method equals of the object target to test for equality. Returns -1 if target is not found.

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to target. Uses the method equals of the object target to test for equality. Returns -1 if target is not found.

(continued)

Some Methods in the Class `ArrayList` (Part 7 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

MEMORY MANAGEMENT (SIZE AND CAPACITY)

```
public boolean isEmpty()
```

Returns true if the calling `ArrayList` is empty (that is, has size 0); otherwise, returns false.

(continued)

Some Methods in the Class `ArrayList` (Part 8 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

```
public int size()
```

Returns the number of elements in the calling `ArrayList`.

```
public void ensureCapacity(int newCapacity)
```

Increases the capacity of the calling `ArrayList`, if necessary, in order to ensure that the `ArrayList` can hold at least `newCapacity` elements. Using `ensureCapacity` can sometimes increase efficiency, but its use is not needed for any other reason.

```
public void trimToSize()
```

Trims the capacity of the calling `ArrayList` to the `ArrayList`'s current size. This method is used to save storage space.

(continued)

Some Methods in the Class `ArrayList` (Part 9 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

MAKE A COPY

```
public Object[] toArray()
```

Returns an array containing all the elements on the list. Preserves the order of the elements.

```
public Type[] toArray(Type[] a)
```

Returns an array containing all the elements on the list. Preserves the order of the elements. *Type* can be any class types. If the list will fit in *a*, the elements are copied to *a* and *a* is returned. Any elements of *a* not needed for list elements are set to `null`. If the list will not fit in *a*, a new array is created.

(As we will discuss in Section 14.2, the correct Java syntax for this method heading is

```
public <Type> Type[] toArray(Type[] a)
```

However, at this point we have not yet explained this kind of type parameter syntax.)

(continued)

Some Methods in the Class `ArrayList` (Part 10 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

```
public Object clone()
```

Returns a shallow copy of the calling `ArrayList`. Warning: The clone is not an independent copy. Subsequent changes to the clone may affect the calling object and vice versa. (See Chapter 5 for a discussion of shallow copy.)

(continued)

Some Methods in the Class `ArrayList` (Part 11 of 11)

Display 14.1 Some Methods in the Class `ArrayList`

EQUALITY

```
public boolean equals(Object other)
```

If `other` is another `ArrayList` (of any base type), then `equals` returns `true` if and only if both `ArrayList`s are of the same size and contain the same list of elements in the same order. (In fact, if `other` is any kind of *list*, then `equals` returns `true` if and only if both the calling `ArrayList` and `other` are of the same size and contain the same list of elements in the same order. *Lists* are discussed in Chapter 16.)

More example code

```
// Note the use of Integer instead of int
public static void main( String[ ] args)
{
    ArrayList<Integer> myInts = new ArrayList<Integer>(25);
    System.out.println( "Size of myInts = " + myInts.size());
    for (int k = 0; k < 10; k++)
        myInts.add( 3 * k );
    myInts.set( 6, 44 );
    myInts.add( 4, 42 );
    myInts.remove( new Integer(99) );
    System.out.println( "Size of myInts = " + myInts.size());
    for (int k = 0; k < myInts.size(); k++)
        System.out.print( myInts.get( k ) + ", " );
    if (myInts.contains( 57 ) ) System.out.println("57 found");
    System.out.println ( "44 found at index " + myInts.indexOf(44) );
}
```

Why are Some Parameters of Type `Base_Type` and Others of type `Object`

- When looking at the methods available in the `ArrayList` class, there appears to be some inconsistency
 - In some cases, when a parameter is naturally an object of the base type, the parameter type is the base type
 - However, in other cases, it is the type `Object`
- This is because the `ArrayList` class implements a number of interfaces, and inherits methods from various ancestor classes
 - These interfaces and ancestor classes specify that certain parameters have type `Object`

The "For Each" Loop

- The **ArrayList** class is an example of a *collection* class
- Starting with version 5.0, Java has added a new kind of for loop called a *for-each* or *enhanced for* loop
 - This kind of loop has been designed to cycle through all the elements in a collection (like an **ArrayList**)

“for-each” example

```
public class ForEach
{
    public static void main(String[ ] args)
    {
        ArrayList<Integer> list = new ArrayList<Integer>;
        list.add( 42 );
        list.add( 57 );
        list.add( 86 );

        // “for each Integer, i, in list”
        for( Integer i : list )
            System.out.println( i );
    }
}

/-- Output ---
42
57
86
```

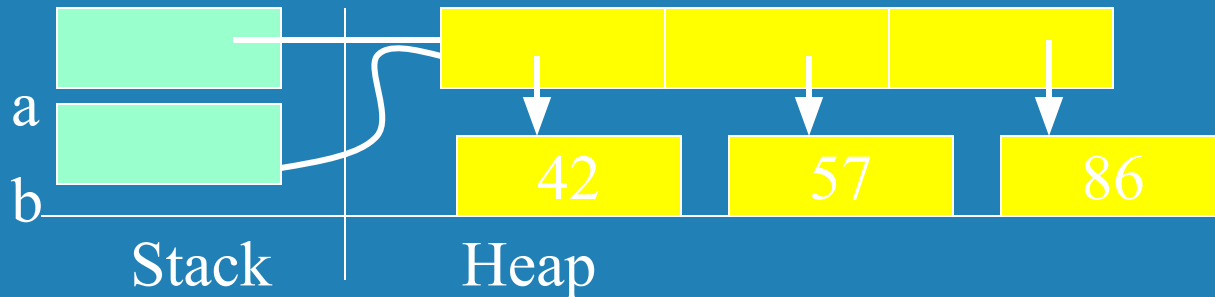
Copying an ArrayList

```
// create an ArrayList of Integers  
ArrayList<Integer> a = new ArrayList<Integer>( );  
a.add(42); a.add(57); a.add(86);
```

❑ Assignment doesn't work

- As we've seen with any object, using assignment just makes two variables refer to the same ArrayList.

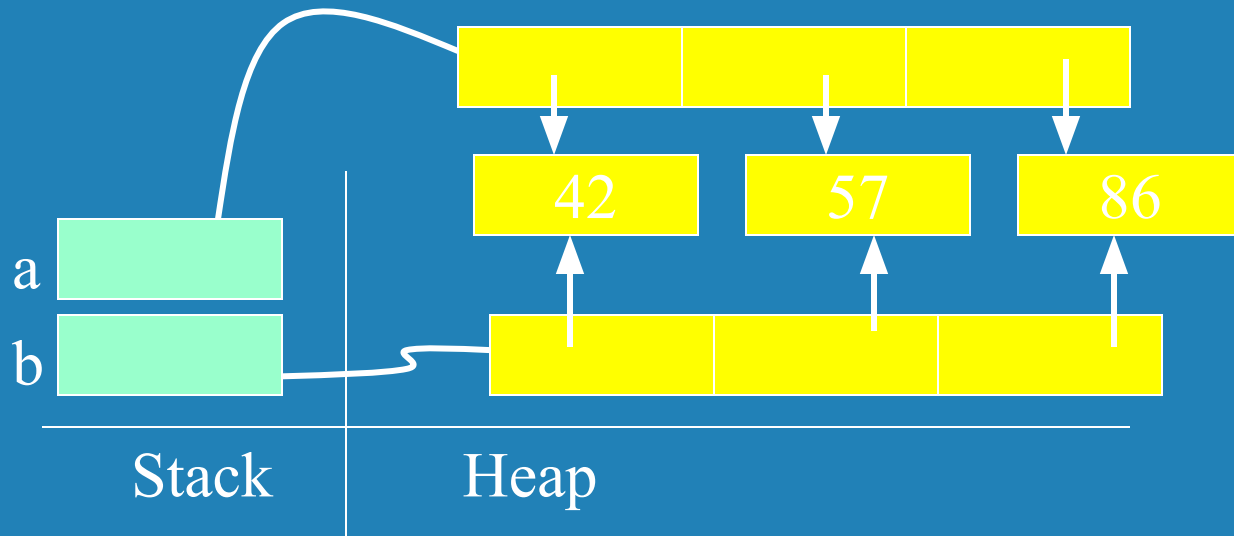
```
ArrayList<Integer> b = a;
```



Copying an ArrayList

ArrayList's clone() method makes a shallow copy

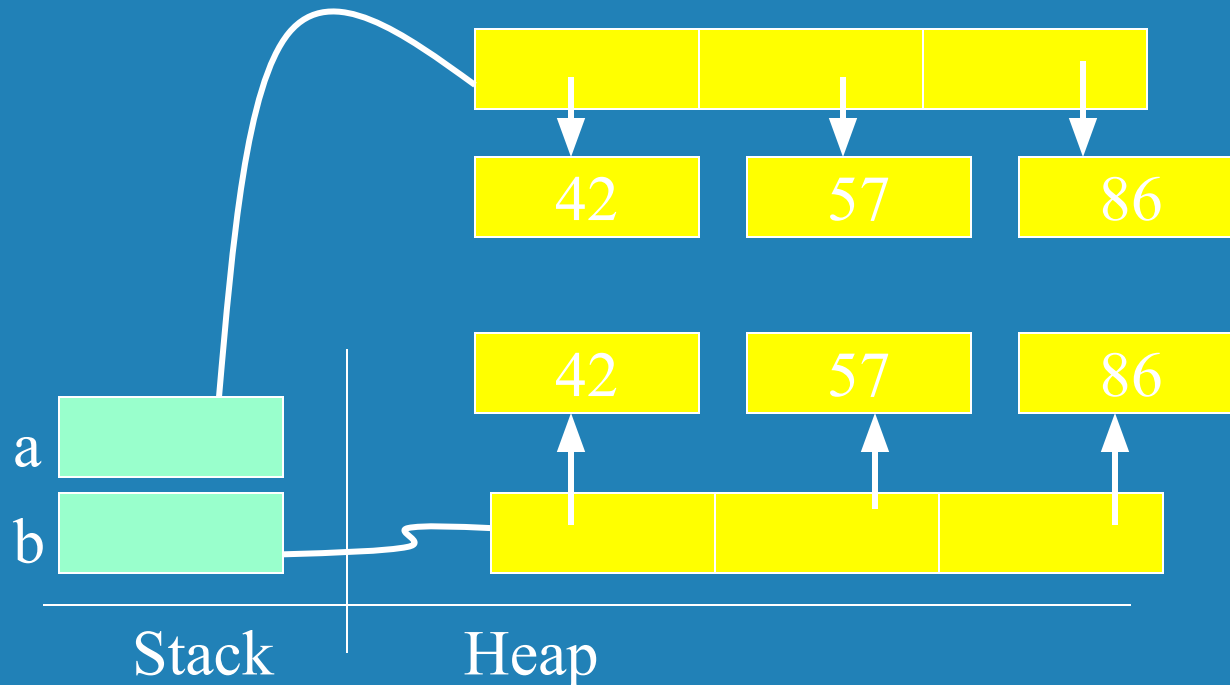
```
ArrayList<Integer> b = a.clone( );
```



Copying an ArrayList

We need to manually make a deep copy

```
ArrayList<Integer> b = a.clone( );  
for( int k = 0; k < b.size( ); k++)  
    b.set(k, a.get(k));
```



ArrayList vs Array

Why use an array instead of an `ArrayList`

1. An `ArrayList` is less efficient than an array
2. `ArrayList` does not have the convenient square bracket notation
3. The base type of an `ArrayList` must be a class type (or other reference type). It cannot be a primitive type. (Although wrappers, auto boxing, and auto unboxing make this a non-issue with Java 5)

ArrayList vs Array

Why use an `ArrayList` instead of an array?

1. Arrays can't grow. Their size is fixed at compile time.
 - ◆ `ArrayList` grows and shrinks as needed while your program is running
2. You need to keep track of the actual number of elements in your array (recall partially filled arrays).
 - ◆ `ArrayList` will do that for you.
3. Arrays have no methods (just `length` instance variable)
 - ◆ `ArrayList` has powerful methods for manipulating the objects within it

The Vector Class

- The Java standard libraries have a class named **Vector** that behaves almost exactly the same as the class **ArrayList**
- In most situations, either class could be used, however the **ArrayList** class is newer (Java 5), and is becoming the preferred class