CS106A, Stanford
Fall, 2004-05

Handout #49
Nick Parlante

ArrayList

So far we have used many sorts of variables, but it has always been true that each variable stores one value at a time – one int or one String or one boolean. The Java ArrayList class can store a group of many objects. This capability will greatly expand what our programs can do. Java has a whole suite of a "Collection" classes that can store groups of objects in various ways. The ArrayList is the most famous and commonly used type of collection, and it is the one we will use the most.

An ArrayList is an object that can store a group of other objects for us and allow us to manipulate those objects one by one. For example, we could use an ArrayList to store all the String names of the pizza toppings offered by a restaurant, or we could store all the URLs that make up a user's favorite sites.

The Java collection classes, including ArrayList, have one major constraint: they can only store pointers to objects, not primitives. So an ArrayList can store pointers to String objects or Color objects, but an ArrayList cannot store a collection of primitives like int or double. This objects-only constraint stems from fundamental aspects of the way Java works, but as a practical matter it is not much of a problem. (Java "arrays" which we will study shortly are an alternative to the ArrayList, and they can store primitives.)

First we will look at a small ArrayList example to see roughly how it works, and then we will look at the real syntax.

ArrayList – Short Version

An ArrayList is just an object, so we will create it like any other object – calling "new" and storing a pointer to the new ArrayList. When first created, the ArrayList is empty – it does not contain any objects. Traditionally, the things stored inside of a collection are called "elements" in the collection. However in Java, collections always store pointers to objects, so we can also use the words "pointers" or "objects" to refer to the elements in a collection.

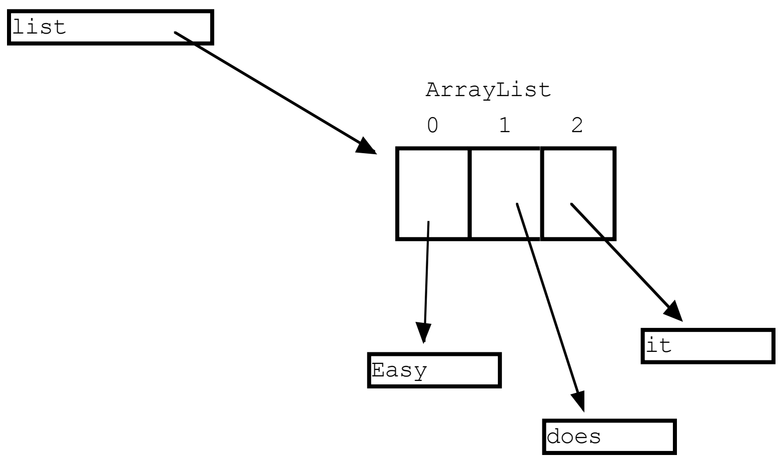
This code creates a new, empty ArrayList...

```
ArrayList list = new ArrayList();  // make a new ArrayList (initially empty)
```

To add an object to the ArrayList, we call the add() method on the ArrayList, passing a pointer to the object we want to store. This code adds pointers to three String objects to the ArrayList...

```
list.add( "Easy" );      // Add three strings to the ArrayList
list.add( "does" );
list.add( "it" );
```

At this point, the ArrayList contains the three pointers, each pointing to a String object...



Index Numbers

How can we identify each of the three elements in the ArrayList? The ArrayList gives each element an "index number". The first element added is called number 0, the next added is number 1, the next is number 2, and so on (the index numbers are shown in the drawing above). The index numbers identify the individual elements, so we can do things with them. This "zero based indexing" scheme is extremely common in Computer Science, so it's worth getting accustomed to it. (Indeed, it is the same numbering scheme used to identify individual chars within a String.)

The ArrayList responds to two methods that allow us to look at the elements individually. The size() method returns the int current number of elements in the ArrayList. The get() method takes an int index argument, and returns the pointer at that index number.

```
int len = list.size();  // size() returns 3 - number of elements

System.out.println( list.get(0) );  // prints "Easy"
System.out.println( list.get(2) );  // prints "it"
```

With the size() method, we can see how many elements there are. With the get() method, we can retrieve any particular element by its index. Since the elements are numbered contiguously starting with 0, the valid index numbers will start with 0 and continue up through size()-1. So if size() returns 3, the valid index numbers are 0, 1, and 2.

So the basic use of an ArrayList reduces to creating it with new ArrayList(), using add() to put elements in, and then using size() and get() to see how many elements there are and get them out.

ArrayList – Long Version
Now we can look at the ArrayList in a little more detail.

public void add(Object element);
To add an object to an ArrayList, we pass a pointer to the object we want to add. This does not **copy** the object being stored. There is just one object, and we have stored a pointer to it in the ArrayList. Indeed, copying objects is very rare in Java. Usually, we have a few objects, and we copy pointers to those objects around.

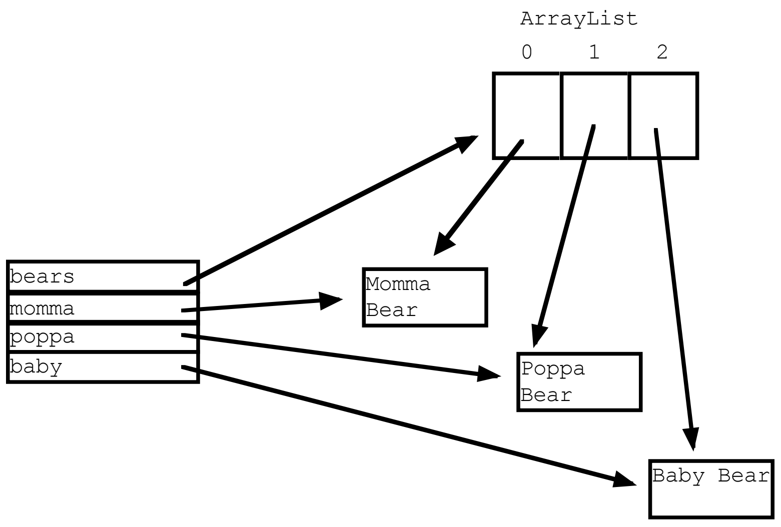
The prototype of add() is: `public void add(Object element);`. The type "Object" means that the argument can be any pointer type – String, or Color, or DRect. We will study the Object type in more detail soon when we look at Java's "inheritance" features. For now, Object works as a generic pointer type that effectively means "any type of pointer".

For example, suppose we have three Bear objects: momma, poppa, and baby. We create a "bears" ArrayList and add pointers to the three bears to the ArrayList...

```
// Create three bears
Bear momma = new Bear();
Bear poppa = new Bear();
Bear baby = new Bear();

// Create an ArrayList, add the three Bear pointers
ArrayList bears = new ArrayList();
bears.add(momma);
bears.add(poppa);
bears.add(baby);
```

The above code produces the memory structure shown below. Notice that the line bears.add(momma) does not add a copy of the momma Bear object to the ArrayList. Instead, it just adds a pointer to momma Bear object to the ArrayList. It is common in Java to have such "shallow" pointers to an object – there is one object with many pointers spread around pointing to it.



public int size();

The size() method returns the int current number of elements in the ArrayList. When the ArrayList is empty, size() returns zero.

public Object get(int index);

The get() method takes an int argument indicating which element we want, and returns that pointer from the ArrayList. The valid index numbers that correspond to actual pointers in the ArrayList are always in the range 0...size()-1, and get() only works correctly given a valid index number. If at runtime we call get() with an invalid index number, such as for the above code, bears.get(100) or bears.get(-2), then Java raises a IndexOutOfBounds exception and stops the program at that point.

The return type of get() is "Object" which means that it can be any type of pointer. However, this will impose a wrinkle on our code. Suppose with the Bear code above, we want to get out the last Bear in the ArrayList. The code for that looks like...

```
// Suppose "bears" is set up with code above

Bear a = bears.get(2);  // NO Very close, but does not compile
```

The above code has the right idea, but there is a problem with the types. As we know, when using an =, Java wants to make sure that the type of variable on the left and the type of value on the right are the same. In this case, the type of the variable is Bear, but the type on the right is Object, since Object is the return type listed in the declaration of the get() method (shown above). The two types are different, so the compiler does not like it.

To fix this situation, we must put in a cast to indicate that we are confident that the type on the right is, in fact, a Bear and so the assignment will be ok...

```
Bear a = (Bear) bears.get(2);  // OK, put in cast to indicate type from get()
```

With the (Bear) cast added, the two sides of the assignment agree and so the compiler is satisfied. This looks a little complex, but it comes down to a simple rule: when calling get(), you must put in a cast to indicate the type of pointer you expect.

At run time, Java will check the type of the pointer, to see if the cast is correct. If the cast is not correct (e.g. if we put in a (String) cast above, which will not match the Bear pointer we are getting), then Java will throw a ClassCastException and stop at that point. So you have to put the cast in your code for each call to get(), but Java actually checks that the cast is correct at run time.

Aside: The "generics" feature added to the most recent version of Java, (Java 5), can do away with the need for the cast on the get(), although generics bring in their own syntactic baggage.

Standard ArrayList For-Loop

It is very common to want to do some operation for all of the elements in an ArrayList. There is a standard, idiomatic way to do that with a for-loop. We loop an int index over the range 0..size()-1. Inside the loop body, we call get(i) to get the element with that index number. It is convenient to call get(i) as the first line of the body, casting the value it returns to the correct type and storing it in a local variable. Then the rest of the loop body can use that variable without worrying about the casting. For example, suppose that Bear object respond to eat(), roar(), and nap() messages. For each bear, we want the bear to do those three operations. The code looks like...

```
// suppose ArrayList "bears" is set up with Bear elements

// 1. Standard loop to iterate over elems in an ArrayList
for (int i=0; i<bears.size(); i++) {
  // 2. For each iteration, get elem number i,
  // cast the value to Bear, and store in a local var "bear"
  Bear bear = (Bear) bears.get(i);

  // 3. Do operations with "bear"
  bear.eat();
  bear.roar();
  bear.nap();
}
```

The above code is a template for any code that wants to loop over all the elements in an ArrayList and do something with each one. The loop corresponds roughly to the phrases in English "for all..." or "for each ...".

Advanced ArrayList Methods

ArrayList and the other standard Java collection classes support many convenient utility methods to operate on their elements. We mention these now for completeness, although our code will mostly use the basic methods: add(), size(), and get().

For the methods below that do comparisons, the ArrayList always uses the equals() method to determine if two objects are the same. The equals() method works correctly for the standard Java classes like String, Color, and so on. The basic methods add(), size() and get() run very fast, no matter how big the collection is. In contrast, some of the methods below must search over the whole collection, and so are potentially much slower. That distinction is noted for each method below.

`void clear();` Removes all the elements from the list, effectively setting its size to 0.

`boolean isEmpty();` True if the list contains no elements. Effectively the same as calling size() and checking if it is == 0, but more readable.

`void set(int index, Object obj);` Replaces the pointer at the given index in the list. The opposite of get().

`boolean contains(Object obj);` Searches the list, returning true if it contains the given object. Uses equals() to compare objects. (somewhat costly)

`int indexOf(Object obj);` Similar to contains(), searches the list from 0..size-1 for the given object. Returns the index where the object is first found, or -1 if not found. Uses equals() to compare objects. (somewhat costly)

`boolean remove(Object obj);` Searches the list for the given object, and removes the first instance of it if found. Objects to the right of the removed object (larger index numbers) all decrease their index numbers by one. Returns true if the object is found and deleted, false otherwise. Uses equals() to compare objects. (somewhat costly)

`void remove(int index);` Similar to remove() above, but takes the index of the element to remove. This version is somewhat faster, since it does not need to first search for the element.