

Software Development (CS2500)

Lecture 22: The Java Library

M.R.C. van Dongen

November 23, 2010

Contents

1	Introduction	1
2	Fixing the Bug	2
2.1	Three Options	2
3	Meet the ArrayList	3
3.1	Adding the Type	4
3.2	Wrapper Classes	5
3.3	Caching	6
3.4	ArrayLists are Iterable	6
3.5	Using the Class	7
4	Comparison	9
5	Packages	10
6	The Java API	11
7	For Wednesday	11

1 Introduction

Today we shall study the Java Library. The Java library provides many useful predefined class code. Specifically, we shall study the `ArrayList` class from the Java Collections. Using `ArrayLists` we shall fix the bug of Lecture 20.

2 Fixing the Bug

At the end of Lecture 20 we were discovered a bug in our SimpleDotCom class. It was caused by the fact that our SimpleDotCom class couldn't remove cells from its locationCells array. The following lists the class and instance variable declarations of the SimpleDotCom class. To allow us to quickly switch to and from testing mode, we've made a little change by introducing a boolean called testing and by letting the construction of the Random object depend on the value of this new variable.

```
public class SimpleDotCom {  
    private static final boolean TESTING = true;  
    private static final int DEFAULT_SEED = 0;  
    private static final Random rand = ( TESTING  
                                        ? new Random( DEFAULT_SEED )  
                                        : new Random( ) );  
  
    private static final int MAX_CELL_VALUE = 6;  
    private static final int CELLS_IN_DOT_COM = 3;  
    private int hits;  
    private final int[] locationCells;  
    ...  
}
```

2.1 Three Options

We have three obvious options to fix the bug

1. The first solution is using a boolean array to record which indices in cellLocations have been hit.
2. Our second option is to use a special value for cell locations which have been hit.

```
private static final int HIT = -1;  
...  
if ( (cell is hit for the first time) ) {  
    cellLocations[ cell ] = HIT;  
}
```

3. Our final solution is neater. We simply remove the cell from the cell locations. We may do this by adding a instance attribute size to count the number of unguessed cell locations. For each first hit of a cell, we simply remove the corresponding cell from the array. If the order of the cells matters this means moving cell locations "down". We may also implement this by moving the last cell to the hit position and by decrementing the size. Of course care should be taken that we may have to re-implement other methods if the new representation of the location cells breaks some invariant which we relied on before. For example, let's assume we deciding to opt for removing a cell by decreasing the size and moving the last cell to the position of the removed cell. Choosing

this implementation would break the invariant that the cells are consecutive, thereby breaking the second and third implementation of the method `findLocation()` from Lecture 20.

```
private int size = CELLS_IN_DOT_COM;
...
if ( (cell is hit) ) {
    size --;
    for (int index = cell; cell != size; index ++ ) {
        cellLocations[ index ] = cellLocations[ index + 1 ];
    }
}
```

Obviously, we have to change the way we deal with hits because we now have to take the `size` attribute into account.

3 Meet the ArrayList

In the previous section we presented three ideas to fix the bug in our `SimpleDotCom` class. Arguably, the last is the prettier, more elegant solution. As it turns out we're fortunate because we don't have to implement the "cell removal" from scratch. It turns out there already is a class that implements the functionality of removing cells from "arrays". The name of the class is `ArrayList`.

An `ArrayList` behaves very much like an array, but it is allowed to grow and shrink. The following describes some of the methods provided by `ArrayLists`.

`void add(Object elem)`: Adds `elem` at the end of this `ArrayList`, thereby increasing the size of the `ArrayList`.

`Object remove(int index)`: Removes object at position `index` from the list and returns the removed object. By removing the object at position `index`, the object with higher indices are moved to the previous index position, thereby filling up the gap. As a result of the `remove()` operation, the size of the array decreases by one.

`boolean remove(Object elem)`: Removes the "first" `Object elem` from the list (if it's in the `ArrayList`). The call returns `true` if and only if an object is removed. As already mentioned, the method removes the "first" object. The meaning of "first" depends on the implementation of the `ArrayList`'s `iterator()` method. This works as follows. The `iterator()` method returns an `Iterator`, which is an object similar to a `Scanner` object. Using the `Iterator` we can iterate over the things in the `ArrayList`. The iteration results in a sequence of objects and the first object, `o`, such that `elem.equals(o)` is removed from the `ArrayList`.

`boolean contains(Object elem)`: Returns `true` if and only if this `ArrayList` contains `elem`.

`boolean isEmpty()`: Returns `true` if and only if this `ArrayList` is empty.

int indexOf(Object elem): Returns -1 if elem is not contained in the ArrayList. Otherwise, it returns the smallest index, i, such that (1) elem == null and get(i) == null, or (2) elem != null and elem.equals(get(i)).¹

int size(): Returns the size of this ArrayList.

Object get(int index): Returns the element at position index in this ArrayList.

Object set(int index, Object o): Set object at position index: This method substitutes o for the original object at position index. The method returns the original object.

3.1 Adding the Type Parameter

ArrayLists are different from normal arrays. They cannot contain primitive type values and may only contain Objects. Let's assume the object type of the objects in the ArrayList is called T. You add the T in angled brackets after the word ArrayList:

```
ArrayList<T> list;
```

Java

Adding the '<T>' adds a level of protection because it tells the compiler that the list should only contain type-T objects (or objects of a subtype of T). You create a new ArrayList in a similar way. Again you add the T in angled brackets after the word ArrayList:

```
list = new ArrayList<T>( );
```

Java

It is possible (and perfectly valid) to omit the <T>. Effectively, this is equivalent to using Object for T: using '<Object>'. First, this makes it impossible for compiler to help you detect errors when adding objects to/ writing objects to the ArrayList.

The following example demonstrates what happens when you don't use the additional type information and start adding objects to the IntArray. The point of the example is that the IntArray which is created in Line 1 is only supposed to contain Child object references. Lines 2 and 3 add some Child objects to the IntArray, which is fine. However, Line 4 adds a GrandParent object reference, which is not a Child object reference. As a result of this last addition, the IntArray now contains objects which it wasn't supposed to contain.

```
ArrayList children = new ArrayList( ); // Hrmmm.
children.add( new Child( "Bart" ) );
children.add( new Child( "Lisa" ) );
children.add( new GrandParent( "Abraham" ) ); // Asking for troubles.
```

Don't Try this at Home

Second, it becomes tedious to get objects from the ArrayList. Specifically, you have to cast most objects to the right type. For example, continuing the previous example, we may write the following.

```
Child child0 = (Child)children.get( 0 ); // Grand but tedious to write.
Child child1 = (Child)children.get( 1 ); // Grand but tedious to write.
Child child2 = (Child)children.get( 2 ); // Run-time error: this is no Child.
```

Don't Try this at Home

¹Make sure you understand why there a distinction between the cases elem == null and elem != null?

Let's examine this example a bit further. Because the compiler only knows that `children` contains `Objects`, we have to explicitly cast these more general `Objects` to `Child` objects before we may assign them to the `Child` variables. The first and second assignments work fine. Unfortunately, the third assignment causes a run-time error because the `GrandParent` object which is stored at position 2 in `children` cannot be cast to a `Child` object.

We shall learn more the advantages of adding the type information when we study the Java *collections* and *generics*. For the moment, it suffices to say that you should *always* add the additional type information.

The following example demonstrates what happens when you do add the additional type information and start adding objects to the `IntArray`. Adding the type information '`<Child>`' tells the Java compiler that the `IntArray` which is created in Line 1 is only supposed to contain `Child` object references. Lines 2 and 3 add some `Child` objects to the `IntArray`, which is fine. However, Line 4 attempts adding `GrandParent` object reference. Since this is not a `Child` object reference, the Java compiler will complain at *compile time*.

```
ArrayList<Child> children = new ArrayList<Child>( ); // Excellent.  
children.add( new Child( "Bart" ) );  
children.add( new Child( "Lisa" ) );  
children.add( new GrandParent( "Abraham" ) ); // Not allowed at compile time.
```

Don't Try this at Home

3.2 Wrapper Classes

We've already seen that `ArrayLists` may only hold objects. As a consequence it is impossible to have primitive type `ArrayLists`. Fortunately, each primitive type has an equivalent `Object` type *wrapper* class. For each primitive type class there is an equivalent object wrapper class which can represent primitive type values. This is probably implemented by having an instance attribute which stores the primitive type value. The wrapper class defines a getter method which lets you get the value of the attribute. However, the wrapper objects are *immutable*, which means you cannot set the value. The names of these wrapper classes are usually formed by turning the first letter of the primitive type in an upper case letter: `Float`, `Double`, `Boolean`, Unfortunately, this convention has two exceptions: the `int` wrapper class is called `Integer` and the `char` wrapper class is called `Character`. Java automatically converts between the primitive types and their object type equivalents.

Autoboxing: The *autoboxing* operation turns a primitive value into its object type.

Unboxing: The *unboxing* operation turns a wrapper class object into its primitive type.

The following is an example of autoboxing. The assignment of the primitive `int` value 2 to the `Integer` variable in Line 2 is a two-stage process. Clearly, assigning a primitive type value to an object reference variable is impossible. Therefore, the primitive type `int` value is converted to an `Integer` in the first stage. This is the autoboxing step. In the second stage the `Integer` is assigned to the variable. The last statement also involves autoboxing because a primitive type `int` value is provided as the argument of `list.add()`, which expects an `Integer`.

```
Integer i1 = new Integer( 1 ); // Create new Integer
Integer i2 = 2; // autoboxing
ArrayList<Integer> list = new ArrayList<Integer>( );
list.add( i1 ); // no autoboxing
list.add( 3 ); // autoboxing
```

Java

The following is an example of unboxing. Basically, this is the converse operation of autoboxing. The values which are wrapped by the boxed type `Integer` are converted to their primitive type equivalents. The statements which involve unboxing are in Lines 2 and 3.

```
Integer integer = new Integer( 3 );
int i1 = integer; // unboxing
int i2 = list.get( 2 ); // unboxing
```

Java

3.3 Caching

Primitive type wrapper classes implement *caching* for a limited number of values. This guarantees that a limited number of deeply wrapper objects are also shallowly equal: This means that for a limited number of objects it is guaranteed that `o1.equals(o2)` if and only if `o1 == o2`. For example, `new Integer(0) == new Integer(0)`. In general this does not always work. For example, `new Integer(666) == new Integer(666)` may not hold. The reason for caching is that it saves memory. In general caching works for “small” primitive values. The following are some values for which caching is guaranteed.

boolean: `true` and `false`.

byte: 0–255.

char: `\u0000`–`\u007f`.

short: -128, -127, ..., 127.

int: -128, -127, ..., 127.

3.4 ArrayLists are Iterable

`ArrayLists` can be used using the enhanced for statement.

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 );
ints.add( new Integer( -1 ) );
ints.add( 2 );
for (Integer i : ints) {
    {use i}
}
```

Java

ArrayLists are really versatile. They have a method `Iterator iterator()` which allows you to traverse the ArrayList from start to end using the `hasNext()`-`next()` mechanism, which we studied for Scanners. Iterators also take type parameters. Moreover, they allow you to delete the “current” element from the iterated collection. The following demonstrates how this may be used to remove all negative values from a given ArrayList consisting of Integers.

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 );
ints.add( 2 );
Iterator<Integer> iter = ints.iterator( );
// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( ); // unboxing
    if (next < 0) {
        iter.remove( );
    }
}
```

More about all this magic when we study Java *collections*, *generics*, and the Iterator design pattern.

3.5 Using the Class

This section shows a possible way to fix the bug in the SimpleDotCom class.

The following shows the class and instance attributes. This time we’re using an ArrayList for our `locationCells` variable. Since we’re storing Integer values, we add the type parameter Integer.

```
private static final int DEFAULT_SEED = 0;
private static final Random rand = ( TESTING
                                     ? new Random( DEFAULT_SEED )
                                     : new Random( ) );
private final static int MAX_CELL_VALUE = 6;
private final static int CELLS_IN_DOT_COM = 3;
private int guesses;
private ArrayList<Integer> locationCells;
```

The constructor is as follows. As should be practice from now on, we add the type parameter Integer to the call to the constructor.

```
public DotCom( ) {
    guesses = 0;
    locationCells = new ArrayList<Integer>( );
    setLocationCells( );
}
```

Changing the method `getResultAsString()` is equally easy.

```

private String getResultAsString( boolean found ) {
    String result;
    if (!found) {
        result = "miss";
    } else if (locationCells.isEmpty( )) {
        result = "kill: you required " + guesses + " guesses";
    } else {
        result = "hit";
    }
    return result;
}

```

Changing `checkYourself()` requires a few changes. First we decide if the given guess corresponds to a cell value in the `ArrayList`. This is done by using the instance method `indexOf()`. The method only returns `-1` if cell is *not* in `locationCells`. Otherwise, it returns the index of the “leftmost” cell in `locationCells` that contains the (autoboxed) value of `cell`. Second, we remove the value `cell` from `locationCells` if it contains the value. This is done using the instance method `remove`: the removal is by the position (index) in `locationCells`.

```

public String checkYourself( String guess ) {
    guesses ++; // increase guesses
    final int cell = Integer.parseInt( guess );
    final int index = locationCells.indexOf( cell ); // autoboxing
    final boolean found = index >= 0;
    if (found) {
        // Remove cell using its index.
        locationCells.remove( index ); // no autoboxing!
    }
    return getResultAsString( found );
}

```

It is important to once more consider the previous solution for `checkYourself`. For example, the instance method `remove` has two possible types: we say that it is *overloaded*. In the first type the method takes an `int` and in the second it takes an `Object`. Since an `int` is used in the previous example, the Java compiler assumes we want to use the first method.

The following is flawed.


```
public String checkYourself( String guess ) {
    guesses ++; // increase guesses
    final int cell = Integer.parseInt( guess );
    final int index = locationCells.indexOf( cell );
    final boolean found = index >= 0;
    if (found) {
        // Doesn't work: int cell is interpreted as index.
        locationCells.remove( cell );
    }
    return getResultAsString( found );
}
```

Don't Try this at Home

The reason why it is flawed is that `cell` is an `int`. Therefore, the compiler assumes that we want to use the method `Object remove(int index)` to remove an object by its index.

However, we can remove the *objects* from the array, so the following should work.

```
public String checkYourself( String guess ) {
    guesses ++; // increase guesses
    final int cell = Integer.parseInt( guess );
    final int index = locationCells.indexOf( cell ); // autoboxing
    final boolean found = index >= 0;
    if (found) {
        final Integer integer = locationCells.get( index );
        locationCells.remove( integer ); // no unboxing
    }
    return getResultAsString( found );
}
```

Java

This time the argument of `remove` is an `Object`. Using this type, the compiler assumes we want to use the method `boolean remove(Object elem)`, which removes its argument (as an object) from the list.

4 Comparison

This section compares ordinary arrays and `ArrayLists`.

- Arrays have a fixed size. Once created, an array's length remains fixed.
- `ArrayLists` grow and shrink. Adding objects to and removing objects from `ArrayLists` affects their size.
 - The call `list.add(Object elem)` adds `elem` to the end of `list` and increments the size of `list`.
 - The call `list.remove(Object elem)` removes the first instance of `elem` from `list` and decrements the size of `list`.

- The call `list.remove(int index)` removes object at position `index` from `list` and decrements the size of `list`.
- Arrays have a special notation to get/set a member: `array[index]`.
- ArrayLists don't have a special notation:
 - The call `list.set(index, elem)` sets the element at position `index` in `list` to `elem`.
 - The call `list.get(index)` returns the element at position `index` in `list`.

5 Packages

The `ArrayList` class is not the only Java library. In this section you will learn a bit more about the library and the packages it provides. The Java API groups classes into *packages*. For example:

- The `javax.swing` package contains lots of GUI-related classes.
- The `java.swing` package contains lots of utility classes.
- The `java.lang` package contains classes which are automatically loaded.

In order to use a class, you need to know the package it's in. There are three ways to use a class from a given package.

1. The classes in `java.lang` are used as per usual. You use the short class name when referring to the class.

```
int rand = (int)(4 * Math.random( ));
```

Java

2. You import a class which is not in `java.lang`. Here you also refer to the class by its usual name.

```
import java.util.ArrayList;
...
ArrayList<Integer> ints = new ArrayList<Integer>( );
```

Java

3. You don't import a class which is not in `java.lang`. This time you refer to the class by its fully qualified name, which is the name you get by writing `<package>.<class>`, where `<package>` is the name of the package and `<class>` is the name of the class.

```
java.util.ArrayList<Integer> ints;
ints = new java.util.ArrayList<Integer>( );
```

Java

There are several advantages to packages.

- Packages help you organise your classes. For example, they help you group related classes into a single package.

- The second advantage is that packages provide *name scope*, which allows you to use the same class name in different packages.

```
package1.Class c1 = new package1.Class( );  
package2.Class c2 = new package2.Class( );
```

Java

- The final advantage of packages is that they provide an extra privacy level on top of `private` and `public`. This security level, which is called package, grants access to all classes in the same package.

Packages are not examinable. You are *not* supposed to implement packages for your assignments.

6 The Java API

There are so many packages and libraries around that may be difficult to learn about them. Given a class, you typically want to know:

- What classes are in the library?
- How to use a given class?

There are two ways to find more about it:

- Read a book. Some Java books provide detailed or less detailed descriptions about packages, classes, and methods and attributes that are provided by the classes.
- Browse the web API documentation.
 - Visit <http://java.sun.com/j2se/1.5.0/docs/api> and read the online documentation.
 - The format of the documentation is standard javadoc.

Alternatively, you may ask a friend or a web forum.

7 For Wednesday

Study the notes and study Chapter 6.