Chapter 1010: ArrayLists

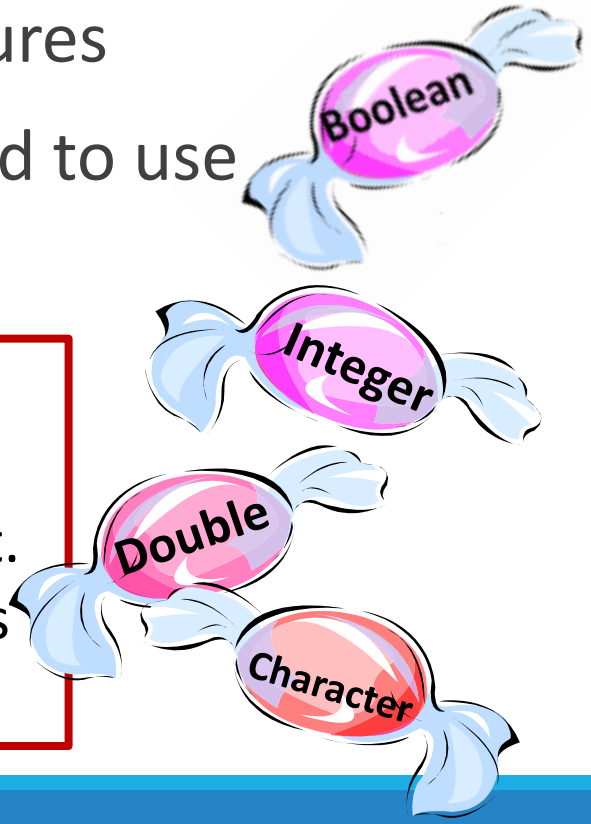
WEEK 2: REVIEW ARRAYS & SEE HOW TO USE AN ARRAY TO IMPLEMENT A NEW DATA STRUCTURE

The List:
A fundamental data structure

- A **DYNAMIC** structure → grows and shrinks as the program executes (more flexible than an array)
- The ***user*** does not care **HOW** the list is implemented
- **ALL** lists will perform the same set of operations
  - ***HOW*** they do this may differ and these differences may affect ***performance***
  - The software developer should be aware of these differences

What do lists store?

- Lists can be used to store values of ***different types***
- This is done by using **generic** structures
- To use **primitive** data types, we need to use **wrapper classes**:



MORE ON THIS LATER, but in brief:
Primitive types are not objects, and thus ***CAN'T*** be used in their native form in an object context. Using the wrapper classes allows primitive types to be used as if they were objects.

The ArrayList class

- As the name implies, the ArrayList class uses an **ARRAY** internally to store list values

SO…How is it different from a regular array?

- An ArrayList ***AUTOMATICALLY RESIZES*** its internal array when there are too many or too few open spaces in the array
- This gives it the flexible performance users expect from a list

ArrayList Performance

- Since internally, the ArrayList class uses an array to implements a list structure, you get the same fast random access you are used to from arrays
- In addition, the ArrayList handles the shifting of values when items are added to or removed from the list, growing and shrinking as necessary
  - Shifting and re-sizing are still time-consuming processes

ArrayList Implementation

- To handle different types, the ArrayList is implemented as a generic class:

  ArrayList<E> where the "E" is short for "Element"

- This "E" gets replaced with whatever type of element the list is to store:

```
ArrayList<String> myList = new ArrayList<String>();
```

NOTE: in Java 7, the second element specification was made unnecessary, so the following should also work:

```
ArrayList<String> myList = new ArrayList<>();
```

Using an ArrayList object

- Once you have constructed an ArrayList, you may use its **class methods**:
  - *add(value)* → adds a value to the end of the list
  - *add(index, value)* → adds a value at a specified index
  - *clear()* → removes all elements from the list
  - *get(index)* → returns the value stored at the given index
  - *remove(index)* → remove the value at the specified index
  - *set(index, value)* → replace the value stored at the given index with the specified value
  - etc. – To see all class methods look at the **List interface**:

http://docs.oracle.com/javase/7/docs/api/java/util/List.html

Adding elements to an ArrayList

- The book has a nice example demonstrating issues to be aware of when using an ArrayList object (p. 658)

Consider the following ArrayList object, **words**, that already contains 6 String values:

words = [four, score, and, seven, years, ago]

You want to add the symbol "**~**" before each value…

How should this be accomplished?

An incorrect attempt:

Be careful when using a for loop to add or remove items

REMEMBER:
An ArrayList object is **DYNAMIC** – it will shift values (and indices) as the for loop is running

```java
import java.util.ArrayList;

public class ArrayListTest {

    public static void main(String[] args){

        ArrayList<String> words = new ArrayList<>();
        words.add("four");
        words.add("score");
        words.add("and");
        words.add("seven");
        words.add("years");
        words.add("ago");

        System.out.println(words);

        // Doesn't work properly
        for(int i = 0; i < words.size(); i++){
            words.add(i, "~");
            if (i > 10)
                break;
        }
        System.out.println(words);
    }
}
```

Problems | @ Javadoc | Declaration | Console

```
<terminated> ArrayListTest [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jul 5, 2014,
[four, score, and, seven, years, ago]
[~, ~, ~, ~, ~, ~, ~, ~, ~, ~, ~, ~, four, score, and, seven, years, ago]
```

WHAT WENT WRONG?

The problem here is that the index of the first word kept changing…

- If the **break** statement had not been included, the for loop would have ***never terminated***
- This happened because the ArrayList is **dynamic**
- With each "**~**" addition, the index of the first word ("four") kept increasing – so index *i* was ***ALWAYS*** the same as the index of the first word
- Looking at the output, you see that the loop kept adding the "**~**" to the front of the word list (at index *i*)
- At the same time, the ***SIZE*** of the list kept increasing
- Consequently, the loop termination condition of ***i* >= words.size()** would **never** have been met

A 2nd (more successful) attempt

```
        // Fixed
        for(int i = 0; i < words.size(); i += 2){
            words.add(i, "~");
        }
        System.out.println(words);
    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ArrayListTest [Java Application] C:\Program Files\Java\jre7\
[~, four, ~, score, ~, and, ~, seven, ~, years, ~, ago]
```

Similar care needs to be taken if you want to remove the "~"s and return words to its original form – but ***be careful***… to remove the alternating unwanted "~" values, a loop index increment value of 1 works, since **index values decrease** as items are removed

Having the loop index increase by 2 after each iteration accounts for the shifting and renumbering of values that occurs after each "~" is added.

A (perhaps) more intuitive approach to the task…

- What if you have the loop work "***backwards***"?

```
// Have the loop run "backwards"
for(int i = words.size()-1; i >= 0; i--){
    words.add(i, "~");
}
System.out.println(words);
```

Problems @ Javadoc Declaration Console

```
<terminated> ArrayListTest [Java Application] C:\Program Files\Java\jre7\bin\
[~, four, ~, score, ~, and, ~, seven, ~, years, ~, ago]
```

This works fine because the part of the list that still needs to be processed is **NOT** being changed.

Using the "For-Each" Loop on a list

- In Chapter 7, you were introduced to the "for-each" loop as a way to iterate over the elements of an array
- These can be used with ArrayLists as well, and can simplify your code:

From THIS…

```
        Integer sum = 0;
        for(int i = 0; i < words.size(); i++){
            String s = words.get(i);
            sum += s.length();
        }
        System.out.println("Sum = " + sum.toString());
    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ArrayListTest [Java Application] C:\Program Files\Java\jre7\bin\javaw.
Sum = 25
```

To THIS…

```
        Integer sum = 0;
        for(String s: words){
            sum += s.length();
        }
        System.out.println("Sum = " + sum.toString());
    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ArrayListTest [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
Sum = 25
```

“For-Each” Loop Rules

- The for-each loop may be used any time you want to ***sequentially*** process values stored in a list
- **HOWEVER**, you **MUST** process **ALL** the items in the list, from first to last
- You **CANNOT MODIFY** a list while you are iterating over it
  - This will generate an **EXCEPTION** (*ConcurrentModificationException*)

Wrapper Classes

Primitive types are not objects, and thus cannot be used in an object context in their native form

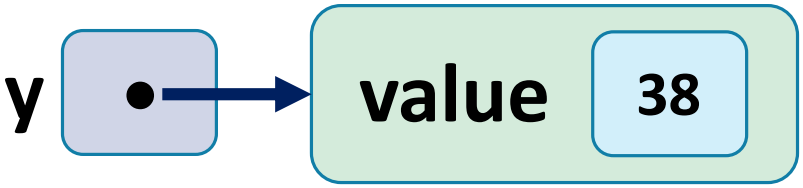
PRIMITIVE TYPE

int x = 38;

- Primitive data is stored directly – the variable x stores the actual value “38”

WRAPPER CLASS OBJECT

Integer y = new Integer(38);



- Objects are stored as references
- The variable y stores a reference to an object that contains the value “38”

Using ArrayList with primitives

- When creating an ArrayList<E>, the "E" has to be a ***reference type*** – it ***CANNOT*** be a primitive

ArrayList<int> **WON'T** work

ArrayList<Integer> **WILL** work

- To make wrapper classes easier to use, Java will convert between primitive and wrapper class values whenever possible
- This is referred to as "***boxing***" (putting a primitive in a wrapper) and "***unboxing***" (removing a primitive from a wrapper)

Boxing and Unboxing

Consider the command: **list.add(42);**

- An int value (42) is being added to the list
- Since Java is aware of the relationship between int and Integer, Java does the int → Integer conversion automatically (***boxing***)

Similarly, consider: **int product = list.get(0) * list.get(1);**

- Java converts the Integer objects obtained from the list to int values (***unboxing***) to do the multiplication, assigning the result to the variable product as an int

The Comparable Interface

- Types that can be sorted have a *natural ordering* of values
  - Putting numbers in order lowest → highest
  - Alphabetizing words
- Any type that has a natural ordering (not all do) should implement the **Comparable** interface:

```
public interface Comparable<T>{
        public int compareTo(T other);
}
```

compareTo Method

- A well-defined *comparison function* is needed to sort values in this way
- The **compareTo** method provides this in every class that implements the Comparable interface
- Defines the relationship between pairs of values:
  - **Less than** (returns a negative number)
  - **Equal to** (returns zero)
  - **Greater than** (returns a positive number)
- It is the **ONLY** method required to implement the interface

The Comparable Interface: Another example of a generic type

- The "T" in Comparable<T> is short for "Type"
- This "T" needs to be replaced when the interface is implemented – more on this soon…
- Two objects ***cannot*** be compared directly using relational operators
- **BUT:** the **result** of any class's **compareTo** method **CAN** be used with a relational operator, since the method returns an ***int value***:

```
if(s1.compareTo(s2) < 0){
    <do something>
}
```

➢ This works to determine whether ***object s1*** is less than ***object s2*** or not

Implementing the Comparable interface

- Many standard Java classes implement the Comparable interface
  - Your own classes can implement the interface as well
- Ex: to have a dates class implement Comparable:

```
Public class CalendarDate implements Comparable<CalendarDate>{

   …

   ***must include the compareTo(CalendarDate other) method***

}
```

You are always comparing pairs of objects of the same type when you implement Comparable, so "T" gets replaced by CalendarDate

Using the compareTo method

Notice how the compareTo method for this class returns more information by not restricting the output to -1, 0, and +1

(the comparison method is also **faster** because the values returned can be ANY negative number, ANY positive number, or zero)

```java
public String toString(){
    return (this.month + "/" + this.day );
}

public int compareTo(CalendarDate other) {
    // TODO Auto-generated method stub
    if (month < other.month){
        return (month - other.month);
    }else{
        return (day - other.day);
    }
}

public static void main(String[] args) {

    ArrayList<CalendarDate> myDates = new ArrayList<>();
    myDates.add(new CalendarDate(8,11));
    myDates.add(new CalendarDate(7,4));
    myDates.add(new CalendarDate(3,14));
    System.out.println("Original List: " + myDates);
    Collections.sort(myDates);
    System.out.println("Sorted List: " + myDates);

    int x = myDates.get(0).compareTo(myDates.get(1));
    int y = myDates.get(1).compareTo(myDates.get(2));
    System.out.println("Result of comparing March and July: " + x);
    System.out.println("Result of comparing July and August: " + y);

}
```

Problems | @ Javadoc | Declaration | Console

```
<terminated> CalendarDate [Java Application] C:\Program Files\Java
Original List: [8/11, 7/4, 3/14]
Sorted List: [3/14, 7/4, 8/11]
Result of comparing March and July: -4
Result of comparing July and August: -1
```

The result of compareTo gives the "distance" between the dates

Major Benefit of implementing the Comparable interface

- As demonstrated in the code on the last slide, a major benefit of implementing the Comparable interface is that it gives you access to built-in utilities like **Collections**
- You can thus use ***Collections.sort*** to sort an ArrayList of CalendarDate objects

Addendum:

- The book has a nice example of using the ArrayList class and the Comparable interface (pp. 678-693)
  - It is worth reading this and making sure you understand the program
- The program uses Scanner objects to read in words from text files
- To ensure that punctuation characters are not being used as parts of words, the program uses a Scanner class method, **useDelimiter**

Continued…

- The useDelimiter method can be called to tell the Scanner object what characters to use when breaking the imput file into tokens (words)
- The method takes a regular expression as an input parameter
  - Regular expressions provide a flexible way to describe patterns of characters
  - More information on regular expressions can be found at: http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Regular Expressions Example

- In this program, only letter characters and apostrophes should be considered "word" characters – all others should be ignored
- This can be accomplished by making all other characters "delimiters"
- To inform the Scanner object of this, the following regular expression can be used:

[^a-zA-Z'']+

The "^" indicates that we are interested in characters NOT in the range provided

Range includes all letters and the apostrophe

The "+" indicates that there may be one or more characters in a row that will be acting as delimiters