CSC 102 Lecture Notes Week 3
Lab and Program Discussion
Program Design
Arrays and ArrayLists

I. Relevant reading.

A. Horstmann chapters 7, 8, and 9.

B. Writeups for Labs 5 and 6 (discussed on Monday).

C. Writeup for Program 2 (discussed on Wednesday).

D. Cited material in writeups.

II. Go over Labs 5 and 6.

A. Here is a summary of the key goals of the labs, as detailed in the writeups:

- get some initial practice using the `ArrayList` class
- understand primitive *wrapper* classes
- get some more practice with the `Scanner` class
- write *overloaded* methods
- start to learn about Java *Generics*

B. Here are a couple sample runs of what Lab 5 can look like; as described in the writeup, the exact format of input and output are up to you.

1. Sample run with correct user input:

```
Enter integers, doubles, bools, or Strings; Enter "quit" when done:
1 2 -10 6 4 2.5 15.6 12.2 true false true hi there
quit
  Minumum integer is: -10
   Average double is: 10.1
  Number of trues is: 2
Number of Strings is: 2
```

2. Here's a sample run that's missing some input, causing the program to throw an exception:

```
Enter integers, doubles, bools, or Strings; Enter "quit" when done:
quit
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
       at java.util.ArrayList.RangeCheck(ArrayList.java:547)
       at java.util.ArrayList.get(ArrayList.java:322)
       at Filter.minimumInt(Filter.java:35)
       at Lab5Driver.main(Lab5Driver.java:44)
```

a. Your solution to Lab 5 can throw exceptions like this, but you should have a look at what the exception says.

b. Coming up soon, you'll learn how to *catch* exceptions like this, so your programs have a more *robust* user interface.

C. The difference between labs 5 and 6 is that lab 5 uses four different ArrayLists to store input data, whereas in lab 6 you use only one ArrayList that can hold four different types of elements. We'll discuss this further below under the subject of java *generics*.

Now on to Arrays and ArrayLists, in Horstmann Chapter 7

III. **Java and C arrays have much in common.**

A. They hold *multiple elements* of the same type.

B. They are a *fixed size*, e.g., `int a[] = new int[10]`

C. The are *indexable*, e.g., `a[i]`.

D. They are *mutable*, e.g., `a[i] = 10`.

IV. **`ArrayLists` are like arrays, but nicer.**

1. They can hold multiple elements of the same type, ***or different types.***
2. They have a ***flexible size***, e.g.,

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

3. They are ***indexable***, but you need to use the `get` method instead of square brackets, e.g.,

```
al.get(i);
```

instead of

```
al[i]
```

4. They are ***mutable***, but you need to use the set method instead of square brackets in an assignment statement, e.g.,

```
al.set(i, 10);
```

instead of

```
al[i] = 10;
```

5. They are ***growable*** using the `add` method, e.g.,`al.add(11)`.

V. **Here's a summary of the primary advantages of `ArrayList` over arrays:**

A. Flexible size.

B. Lots of useful methods.

C. Enhanced `for` loop, e.g.,

```
for (int i : al) { ... }
```

VI. **Let's look at an example, in *`102/examples/ArraysAndArrayLists.java`***
which we'll walk through during lecture.

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

/****
 *
 * This class illustrates some of the basic ideas for arrays and ArrayLists.
 * In 102, you'll primarily be using ArrayLists instead of arrays, but use of
 * arrays may be convenient in some cases.  NOTE: In labs and programming
 * assignments where it says you must use an ArrayList, using an plain array
 * will not do.
 */
public class ArraysAndArrayLists {

    public static void main(String[] args) {
```

```
        // Allocate a 10-element array.
        int a[] = new int[10];

        // Allocate a flexible-size ArrayList.
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Assign the values 0 through 90 to both the array and ArrayList.
        for (int i = 0; i < 10; i++) {
            a[i] = i * 10;
            al.add(i * 10);
        }

        // Print out the elements of the array, using a standard for loop.
        for (int i = 0; i < 10; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();

        // Print out the elements of the ArrayList, using standard for loop.
        for (int i = 0; i < 10; i++) {
            System.out.print(al.get(i) + " ");
        }
        System.out.println();

        // Increment each element of the array and ArrayList by 1.
        for (int i = 0; i < 10; i++) {
            a[i]++;
            al.set(i, al.get(i) + 1);
        }

        //
        // Print out the elements of the array and ArrayList in different ways.
        //

        // Use the Arrays.toString library method on the array.
        System.out.println(Arrays.toString(a));

        // Use (indirectly) the ArrayList.toString method.
        System.out.println(al);

        // Use the specialized form of for loop on ArrayLists.
        for (int i : al) {
            System.out.print(i + " ");
        }
        System.out.println();

        // Try to print the array directly; what's going on here?
        System.out.println(a + "   -- Say what?");
    }
}
```

VII. **Java Generics**

A. Consider closely the ArrayList definition in the preceding example:

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

B. The angle brackets around `Integer` denote a ***generic*** definition.

C. This means that the type of an `ArrayList`'s is *generic*, that is, an ArrayList can hold any type of object.

D. E.g., we can have `ArrayLists` of `Integers` or `Doubles` or ... .

E. The most generic type of `ArrayList` holds `Object`s, e.g.,

```
ArrayList<Object> al = new ArrayList<Object>();
```

F. The transition from lab 5 to lab 6 goes from using four type-specific `ArrayLists` to one *fully generic* `ArrayList` of `Object`s.
   1. The lab 6 writeup explains the details of how to do this.
   2. Section 7.2 of the book has further discussion of declaring generic ArrayLists.

VIII. **Wrapper classes (Section 7.3 of the book).**

A. The rules of Java say that ArrayLists can only hold objects, not primitive types.
   1. This means that the following definition results in a compiler error:

```
ArrayList<int> al;
```

   2. A smarter Java compiler might be able to cope with a definition like this.
      a. An in-depth explanation of why it can’t is beyond the scope of CSC 102.
      b. If you’re curious, you can Google around for some discussion of the subject; for example, try the Google search *"why is java stupid about primitive types in collections"*.

B. To get around the problem of no primitives in ArrayLists, the primitive types have "wrapper" classes.

C. For example, the `Integer` class wraps the primitive `int`.

D. Such wrapper classes are used in ArrayLists and other Java collections.

E. In summary, you always declare and create ArrayLists like this

```
ArrayList<Integer> = new ArrayList<Integer>
```

as opposed to this

```
ArrayList<int> = new ArrayList<int>
```

F. To make life a little less painful in dealing with wrapper classes, Java version 5 introduced features called "auto-boxing" and "unboxing".
   1. For example, you can add what looks like a primitive ***`int`*** value of 10 like this

```
al.add(10);
```

   and Java will "auto-box" it to this

```
al.add(new Integer(10));
```

   2. "Unboxing" is also automatic, as in

```
int i = al.get(x);
```

   being equivalent to

```
int i = al.get(x).intValue();
```

G. Section 7.3 of the book discusses this subject further.

IX. **Introduction to Java interfaces (Chapter 9).**

A. An interface defines a form of completely abstract class.

B. The interface definition has just methods, with no data fields.

C. All interface methods are *fully abstract*.

1. They have names and signatures, but no implementations
2. A method *signature* consists of
   a. The types of its parameters, in the order they are declared.
   b. The return type.
   c. E.g., the signature of

   ```
   double methodX(int i, String s, boolean b);
   ```

   is

   ```
   (int, String, boolean) -> double
   ```

   which reads "A method of `int` , `String` , `boolean` returning `double`".
3. The declaration of an interface method ends with just a "`;`", not a body of code in "`{ . . . }`".

X. What interfaces are good for.

A. The primary use for an interface is defining common behavior for classes; the upcoming example will illustrate this idea.

B. Interfaces can be particularly useful when the behavior only needs a few methods to define; this will be illustrated by some of the Java library interfaces that we'll be looking at in the coming weeks.

C. Having common behavior defined in an interface allows classes that use an interface to work to deal easily with different types of data; this is called ***polymorphism***, and upcoming examples will show how it's useful.

XI. A good example of where an interface could be useful is provided by the Program 3 Shape interface.

A. I want to write a drawing program, that will display geometric shapes on a screen; these are shapes like rectangles, circles, etc.

B. So far in my 102 programming examples, I've figured out how to code a rectangle, so how about I define a drawing like this:

```
public class Drawing {
  ArrayList<Rectangle> canvas;
  public static void main(...) {
     // Draw some stuff on the canvas
  }
}
```

But this is really boring, since all I can draw is rectangles.

C. What I actually need is something like this:

```
public class Drawing {
  ArrayList<Shape> canvas;
  public static void main(...) {
     // Draw some stuff on the canvas
  }
}
```

D. So it looks like I could use a `Shape` class.

E. So, what do geometric shapes have in common that will go in this `Shape` class?
   1. Number of points? *(not really)*

2. Moving around? *(based on points*)
3. Sizes? *(computed differently)*

F. Shapes could be a class, but
1. They may not have any common data.
2. They probably have different method implementations.

G. Enter interfaces.

XII. **Comparison of classes and interfaces.**

A. Here's what a `Shape` interface looks like, and how it can be implemented by a `Rectangle` class:

```
public interface Shape {
    public void move(Point delta);
    public double getArea();
    . . .   // more later
}

public class Rectangle implements Shape {
    int x,y,height,width;
    public void move(Point delta) {
        ...
    }
    public double getArea() {
        ...
    }
}
```

B. Suppose instead of defining `Shape` as an interface, we defined it as a class, like this:

```
public class Shape {
    // Leave out data fields
    public void move(Point delta) {
        // no default implementation
    }
    public double getArea() {
        return 0;    // pretty useless
    }
    . . .   // maybe more later
}

public class Rectangle extends Shape {
    int x,y,height,width;
    public void move(Point delta) {
        ...
    }
    public double getArea() {
        ...
    }
}
```

C. The comments in the class definition suggest why an interface definition may be the better choice for `Shape`s than a class.

1. "May be" a better choice will be a subject of further discussion next week.
2. In fact, the subject of Program 3 will specifically address the issue of choosing between an interface or class definition of shapes.

XIII. **Summary of what it takes to implement an interface.**

A. Use the keyword `implements`.

B. Implement ***all*** interface methods.

C. Declare interface methods `public`.

D. Method names and signatures in interface and implementation must ***exactly match***.

XIV. **Polymorphism (Section 9.3).**

A. It's Greek for "multiple shapes".

B. In a program, it means a method can take different "shapes", i.e., types, of data.

C. In the example above, the `Drawing` class has an ArrayList of `Shape` for its drawing canvas.

1. It can move shapes around on the canvas without knowing what particular type of object it's moving.
2. This is possible because the `Shape` interface requires that all of its implementing classes provide their own definition of the `move` method.
3. That is, all of the methods of a `Shape` are ***polymorphic***.
4. This is a powerful feature of interfaces about which we'll have more to say in upcoming weeks.

Now on to a Bit of Program Design, from Horstmann Chapter 8

XV. **Introduction to software design.**

A. Design is an *abstraction* of the implementation.

B. Abstraction means "leave out some details".

C. **We'll focus on three levels of design in CSC 102**

1. ***Detailed*** -- leave out method bodies, i.e., all of the code between the curly braces that implement methods.
2. ***Intermediate*** -- leave out private data and private methods.
3. ***High-level*** -- leave out all methods entirely, i.e., just design with the names of classes.

D. **Design can be expressed in a number of different languages.**

1. The Java code itself, with some details omitted.
2. Javadoc web pages, which are generated from the code.
3. Diagrams in UML, the Unified Modeling Language.

XVI. **A Very Brief Introduction to UML.**

A. UML is a graphic form of design.

B. It can be convenient to show "the big picture" for a program.

C. For 102, we'll use a very small subset of UML.

D. Elements of a UML class diagram are the following:

1. a one-part box, containing the name of a class
2. a three-part box, containing:
   a. class name on the top
   b. data fields next
   c. methods on the bottom
3. Connection lines of different forms, including
   a. solid lines with triangular arrows show ***inheritance*** (which topic will cover in upcoming lectures)
   b. dashed lines with open arrows show ***dependencies***

E. Some UML details:

1. Data are OK with or without field names.
2. Methods are OK with or without signatures.
3. Class diagrams versus object diagrams
   a. Class names are underlined in object diagram.
   b. Also, object diagrams show the actual values of data fields.
   c. See the example just below, plus examples in the book.
4. There are further details in the book, which we will discuss next week.
   a. In practice, UML usage varies from person to person.
   b. This is OK, since UML allows for some flexibility.
   c. Next week we'll clarify the UML notation to be used in 102, including the small additions we're making to the book's notation.

XVII. **A Very Small UML Example.**

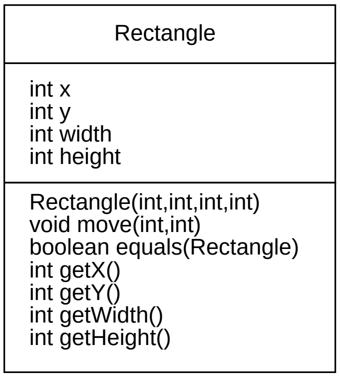
A. Consider the class `Rectangle.java`.

1. We've seen the code already:

```
/****
 *
 * A simple Java program that defines a rectangle data structure
 * and methods that operate on rectangles.
 *
 */
public class Rectangle {
    int x;
    int y;
    int width;
    int height;

    Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    void move(int x_increment, int y_increment) {
        x = x + x_increment;
        y = y + y_increment;
    }

    boolean equals(Rectangle r) {
```

```
            return x == r.x &&
                   y == r.y &&
                   width == r.width &&
                   height == r.height;
        }

        public int getX() {
           return x;
        }

        public int getY() {
           return y;
        }

        public int getWidth() {
           return width;
        }

        public int getHeight() {
           return height;
        }
    }
```

2. We’ve also seen the Javadoc.
3. Here’s a UML class diagram:



4. Here’s a UML object diagram: