CS 302: Introduction to Programming in Java

Lecture 13

Review

- What is the 3-step processing for using Objects (think Scanner and Random)?
- Do objects use static methods or non-static (how do you know)?
- What value does x[0] have if:

String[] x = new String[5];

- How are angle brackets used in ArrayLists?
- What are Wrapper Classes?

Using an ArrayList Object

ArrayList<type> variableName = new ArrayList<type>();

↑

Must be a reference type (ex.
String) – cannot be primitive
(ex. int, double, boolean, char)

variableName.size() - returns the size of the ArrayList as an int

variableName.add(element) - appends element to the end of the list and automatically increases its size

variableName.set(i, element)

0 <= i < variableName.size() - sets variableName → i= element

Reading Input

```
ArrayList<Double> testScores = new ArrayList<Double>();
while (in.hasNextDouble())
{
   testScores.add(in.nextDouble());
}
```

Length with Arrays, ArrayLists, and Strings

- Stings -> stringName.length();
- Arrays -> arrayName.length;
- ArrayLists -> arrayList.size();

Selection Sort

- Write a method to sort an ArrayList of Integers using Selection Sort:

```
public static void selectionSort(ArrayList<Integer> data)
```

- Selection Sort (basically what humans usually do):
  - For each index in the array:
    - Find the current smallest element from [index...end]
    - Swap its value with the element currently in index

Object Oriented Programming (OOP)

- Programming style invented to enhance code management and maintainability
- Basic Idea: split code into "objects"
  - Each "object" represents a discrete thing or idea (ex. a cash register, a phonebook, a robot)
  - Each object has its own set of methods for creating an instance of an object and using the object
- Code thus becomes an interaction of objects

Procedural Programming vs OOP

- Procedural Programming
  - Goal – break code down into sub-routines and variables
  - Pro – can quickly address the problem at hand
  - Con – Difficult to maintain and adapt to new problems
  - Real-world counterpart: a cooking recipe

- OOP
  - Goal – break code into discrete objects that interact with eachother
  - Pro – easy to maintain, can reduce code volumn
  - Con – Can take longer to create
  - Real-world counterpart: a play

Objects in Java

- What objects have we already worked with?
- Implementing objects
  - We do NOT code up individual objects
  - Instead we define classes
    - Each class represents a generic object (i.e. the String class defines the behaviour for all Strings)
  - When we want to use an object we create a new instance of that object from the class code (use the "new" keyword)
  - Ex. Random randGen = new Random();

Instance Methods

- Each object must define its own methods (e.g. a cash register object would have a method to add prices)
- Methods that can be invoked on objects = Instance Methods (non-static)
- Each object must have a special type of instance method: a Constructor
- A constructor creates a new instance of the object (it is called when you instantiate a new object of this type)
- Ex. Random randGen = new Random();

↑ Constructor

Public Interface and Encapsulation

- Idea: Can treat objects as black boxes just like we were treating methods
- Ex. I don't need to know how the Random constructor works or how the rand.nextInt() method is implemented as long as I know how to call it
- Real-world example: I don't need to know how the electronics in a computer work in order to use the computer
- User only needs to know the **public interface** = how to interact with the object
- **Encapsulation** = Using a public interface to hide implementation details

General Class Design

- Your class code defines how every object of this type will work (a blueprint for the object)

```
public class Phonebook
{
   private data;
   public Phonebook() //Constructor
   { }
   //Methods someone using a phonebook would need
   public String getPhoneNumber(String name)
   { }
   public String addNumber(String name)
   { }
}
```

Someone using a phonebook object doesn't need to know how these methods are implemented, only what arguments they expect and what they return

Accessors and Mutators

- Most instance methods can be divided into 2 types: Accessors and Mutators (also called "getters" and "setters")
- Accessors = methods that access data but do not change the object
- Mutators = methods that modfiy the object

Instance Variables

- In addition to having its own methods, most objects need to store data in some way
- Ex. a phonebook would need to store all the names and numbers in the phonebook
- To do this we user Instance Variables = variables defined within a class
- Ex. Phonebook might have 2 ArrayLists – one for names and one for numbers
- Instance variables are declared within the class but outside any methods – this means any method in the class can use these variables

Instance Variables Example

```
public class Phonebook
{
  private ArrayList<String> names;
  private ArrayList<String> numbers;
  //Constructor and other methods follow
}
```

- Each instance of a phonebook will have its own seperate copy of names and numbers
- Note the "private" declaration
- How would accessors and mutators work?

Implementing Instance Methods

- Similar to implementing the static methods we have done before
- Constructors intialize the instance variables, do not return anything, and do not have a type
- Constructors must have the same name as the class (object)
- Can have multiple constructors each that takes in different parameters (ex. Random rand = new Random() vs Random rand = new Random(seed)

Practical Example

- How can we implement a bank account object?
- What private instance data will we need?
- What sort of Accessors and Mutators will we need?
- What will the constructor look like?