CSE 142

Introduction to Collections – ArrayLists

Outline for Today

- Collections of data
- APIs
- ArrayLists
- Technicalities
  - Objects
  - casts
  - reference vs primitive types

Collections in the Real World

- Think about:
  - words in a dictionary
  - list of students in a class
  - deck of cards
  - books in a library
  - MP3 files on a computer
- These things are all ***collections***
- They contain multiple instances of like objects
- Some collections are ***ordered***, others are ***unordered***

Collections in Some Familiar Models

- Bank, BankAccount
- Student, Registration
- Airplan factory
- Cell Phone, Media Player

Some Common Types of Collections

- Collections may or ordered or unordered
- Some collections are "sets"
  - no inherent order
  - duplicate elements not allowed
- A very common collection type is a *list*
  - Elements in a list are in a definite order, one after another

Collections and Libraries in Java

- The Java language does not have special keywords or syntax for collections
- Collections and lists are available in Java programs through *class libraries* that are part of every Java implementation
- There are standard Java class libraries for dozens or hundreds of purposes
  - Math
  - Graphics
  - Networking
  - Files
  - Collections
  - etc., etc.

More About APIs

- The phrase API (application programming interface) is commonly used to designate a set of classes and methods
- To be an effective Java programmer, you must use APIs!
- Must learn *how* to use them
  - What to expect
  - Requirements and conventions of programming
  - Conventions of documentation
- Must learn *specifics* of particular APIs
  - Which classes and methods are available
  - The internal model of the application
- A long-range goal of 142/143 is to make you confident about using APIs

An Ordered Collection in Java: ArrayList

- ArrayList is a Java class whose instances store an ordered collection of things
- ArrayList is one of a number of standard Java library classes for collections
- You can add objects to an ArrayList object and get them back out
- No limit to the length of a list

Some ArrayList Methods

- The specification for ArrayList tells us what methods are available. A few of the methods:

```
public class ArrayList
    // Create an empty collection
    public ArrayList( );

    // Add the given object to the end of this collection
    public void add(Object o);

    // Return the size of this collection
    public int size( );
    ...
}
```

- New: Object type – means any kind of object at all

Using ArrayLists

- Creating a list: ArrayList is a class, and we need an instance of the class (object) to store data:

```
ArrayList names = new ArrayList ( );
```

- Adding things:

```
names.add("Billy");
names.add("Susan");
names.add("Frodo");
```

- Getting the size:

```
int numberOfNames = names.size( );
```

- If you try typing the above into Dr. Java... it won't (quite) work!

The *import* Statement

- ArrayList is *not* a keyword of Java
- Any classes not defined in your own program must be *imported*
- The *import* statement tells the compiler which library or external classes you want to use
- ArrayList is in a "package" called *java.util*
- Write *import java.util.*;* to use classes of the *java.util* package
- All *import* statements must be at the beginning of the .java file
  - In DrJava's interaction window, you can type them anytime

Drawing Diagrams

- Diagrams are useful for
  - Describing
  - Communicating
  - Understanding
- Many types of diagrams are possible for various situations
- In CSE14x, we often draw a diagram to show the relationships between names and objects
- These are "dynamic" in the sense that
  - they depict the program at run-time, not at compile-time
  - they capture one particular instant of execution
  - they focus on the relationship between objects, not classes
  - Such a diagram can change after each statement execution, or even during statement execution

Groundrules

- Each object is a blob; each blob is an object
- Arrows go from names to objects
- Local variable names are freefloating
- Instance variable names are written inside their object blob
- Primitive values are not blobs
  - Write primitive values close to their names
  - Some people use a small box, others use an arrow

Drawing Dynamic Status Diagrams

- DO...
- Draw a separate blob for each object
  - One blob, one object
- Label each blob with its type
- Write each local variable name floating free
- Draw an arrow from a name to the object it refers to
- Draw a rectangle to show a class (if needed)
- Write instance variable names inside their class blob
  - or free-floating if the blob is not drawn
  - Remember that Strings are objects

- DON'T...
- draw one blob inside another – ever!
- complicate the drawing with unused or unnecessary details
- draw arrows between blobs
- draw arrows between names
- draw blobs for primitive values
- write variable names inside boxes or as labels to boxes

ArrayList Diagrams

- The indexes of an ArrayList are a form of name
- Inside the blob for an ArrayList object, write the indexes in a row
  - Show only the indexes that are actually in use (i.e. which are not currently out of bounds)
- Draw an arrow from each index to the object it refers to
- PS We will elaborate this picture later in the course, after studying arrays.

More ArrayList Methods

- Here's more of its interface:

```
public class ArrayList {
    ...
    // Return the object at the given index (numbered starting from 0, not 1!).
    // Raise an exception if index is out-of-bounds.
    public Object get(int index);

    // Change the object at the given index (starting from 0) to be newElement.
    // Raise an exception if index isn't in bounds.
    // Return the element that used to be there.
    public Object set(int index, Object newElement);
}
```

Using Indexes with ArrayLists

- ArrayLists provide *indexed* access. We can ask for a particular item in the list by its position or *index* number
- The first item is at index 0, the second at index 1, and the last item is at index *n*-1 (where *n* is the size).

```
ArrayList names = new ArrayList ( );
names.add("Billy");
names.add("Susan");
```

- Java expressions:

```
names.get(0)
names.get(1)
```


A Problem

- Let's say we want to get something out of an ArrayList and assign it to a variable
- We might write the following:

```
String name = names.get(0);
System.out.println("The first name is " + name);
```

- But Java complains about the green line: "incompatible types: found: Object, required: String"
  (DrJava's interactions window allows this without complaining, even though it's not legal in regular Java)
- Why? [Hint: look at the interface of the get method]

Problem: *Object*

- The return type of method *get*() is *Object*.
- Think of *Object* as Java's way of saying "any type".
- All classes in Java (including the ones we write) have an "is-a" relationship to *Object*. In other words:
  - every String is an Object
  - every Rectangle is an Object
  - every ArrayList is an Object
- The reverse is not generally true!
  - every *Object* is not necessarily a String

Making False Claims

- Looks weird, but is legal...

```
Object someObject = new Soap(. . .);
```

  ... because every Soap is an *Object*.
- In our example:

```
String name = names.get(0);
System.out.println("The first name is " + name);
```

- We are claiming that an Object (the result of get) is a String, which is not necessarily true!
  - What if we passed an ArrayList of Soap to printFirstName?

Making Promises: Casting

- It looks like we're stuck. We can add things to the collection, but we can't get them back out!
- The solution is to make a promise
  - Say that *we* know that we've only placed String objects into the ArrayList.
  - We can *promise* the compiler that the thing coming back out of the ArrayList is actually a String

```
String name = (String)names.get(0);
System.out.println("The first name is " + name);
```

  - This is (another use of) a ***cast***

Casting (Review)

- Pattern:

  (<class-name>)<expression>

  - Example:

    String name = (String)names.get(0);

- Casting does *not* change the object or the type of the object.
- It is a promise that the object really is of the stated type.
- Casting also used for conversions, as we've seen.

  (int) 3.1415927

Miscasting

- We can abuse casting, but it will be caught at runtime.

```
String name = (String)names.get(0);
System.out.println("The first name is " + name);

Rectangle box = (BankAccount)names.get(0);    // Run time error!!
System.out.println("The length is " + box.getLength());
```

- An error called a "class cast exception" occurs if a promise is broken.
  - Footnote: “Exceptions” are one way that programs signal that something unexpected or undesirable has occured (CSE143 topic)

Reference vs. Primitive Types

- A few Java types are ***primitive***

  int, double, char, boolean, and a few other numeric types (see textbook)
  - Are atomic chunks, with no parts (i.e., no instance variables)
  - Exist without having to be allocated with new
  - Cannot receive messages (i.e., do not have methods) but can be arguments of messages and unary and binary operators
- All others are ***reference types***

  Rectangle, BankAccount, Color, String, etc.
  - Instances of some class
  - Created by new
  - Can have instance variables and methods
  - All are special cases of the generic type “Object”

When Does the Distinction Matter?

- One place: when putting values in collections

```
ArrayList list = new ArrayList( );
list.add(5);                    // error: int isn't an Object
```

- The way ArrayList is defined, only objects can be added to the list
  - Reminder: true objects have a "reference type"

Using Wrapper Classes

- There is a solution
- If we really need to put a primitive value in an ArrayList: create a *wrapper* object containing the primitive value.
- There is a wrapper class for each primitive type, e.g. Integer for int, Double for double, etc.

```
ArrayList list = new ArrayList( );
Integer five = new Integer(5);      // create an Integer object with a 5 in it
list.add(five);                     // ok: Integer is an Object
...
Integer firstElem = (Integer) list.get(0);  // promise that the Object is an Integer
int v = firstElem.intValue( );      // extract the int value from the Integer object
```


Summary

- Collections: Many kinds
  - Common in computer programs
  - Often correspond to collections of objects in the real world
- A simple collection: ArrayList
  - Sequential, ordered collection
  - Part of the *java.util* package of classes
  - Many methods: add, get, size, isEmpty, ... (see Sun Java Docs)
  - import. java.util.*; to access
- Casts
  - Often needed to specify actual type of object retrieved from a collection (since collection can hold any kind of object)
- Primitive vs. reference types: need to place primitive values in wrapper objects if we want to store them in a collection