**ECS140A-F16-06** **October 18, 2016**

# ASSIGNMENT 4: Java object oriented program

Due: October 27, 2016

## Overview

The purpose of this assignment is for you to gain experience with object oriented programming in Java: in particular, the ideas of inheritance, dynamic binding (virtual methods), overriding and overloading methods, iterators (aka generators). It will also give you some exposure to how some programming language features are implemented, i.e., what happens "behind the scenes" when you execute a program. Although no language provides the exact features in this assignment, a number of languages (e.g., Awk, Icon, LISP, CLU) do provide similar features. (In fact, you could define a little language, using techniques from first program, that would contain features like those in this assignment.)

Your program needs to provide several abstractions. The first abstraction is an *Element*, of which there are three kinds: *EBoolean*, *EInteger*, and *EString*. The second abstraction is an *EList* of *Element*; associated with the *EList* abstraction is *EListIt*, which iterates over a list. Details of these are given below in each part of this assignment.

N.B., you are restricted as to how you write your code. See "Details" before you start coding.

## Part 0: Basic Derivation

You will be given an *Element* class. From it, derive *EBoolean*, *EInteger*, and *EString* classes whose constructors' signatures are:

```
EBoolean( int v )
EInteger( int v )
EString( String v )
```

For an *EBoolean*, a non-zero *v* indicates true and a zero *v* indicates false; for an *EInteger*, *v* gives the integer's value. For an *EString*, *v* specifies the string initial value; however, a string can contain at most 24 characters, so any excess characters are (silently) truncated.

In each class, provide code for the class's constructor and override the *toString* method. (*toString* comes from *Object*, so it doesn't need to be declared in *Element.*) *toString* for an *EBoolean* gives the string “$true$” or “$false$”; *toString* for an *EInteger* gives the string representation of the integer; *toString* for an *EString* gives the string. Return only the values; do not add extra newlines or white space.

In this part or a later part, you might find it useful to convert an integer to a string. Use Java's

```
String.valueOf(num); // num is an int
```

or simply

```
""+num; // num is an int
```

In defining the data fields for these classes, use **protected** rather than **private**, as explained further in Part 3. (Or if you prefer, you can define the data fields as **private** and provide **protected** access methods for them.) Only the constructor and the *toString* method should be public.

## Part 1: The hash method

Define a method, *hash*, that returns the hash of an *Element*. More specifically, define an abstract *hash* method in the *Element* class with signature

public abstract int hash()

Override the *hash* method in each element class. *hash* for an *EBoolean* gives 1 for true and 0 for false; *hash* for an *EInteger* gives mod base 17 of the absolute value of the integer; *hash* for an *EString* gives mod base 17 of the length of the string (this hash function isn't particularly good, but it serves our simple purposes for this assignment).

## Part 2: Counting Instances

Add to the *Element* class the following method:

public static int getCount()

It returns the number of *Element* objects that have been created, i.e., the number of times the *Element* constructor has been invoked. Similarly, add *getCount* methods to the *EBoolean*, *EInteger*, and *EString* classes. *getCount* within *EBoolean* returns the number of *EBoolean* objects that have been created. (Similarly for *EInteger* and *EString*.)

## Part 3: Polymorphic Overloaded Plus

The following table shows the type and value of the result of *plus*(*a*, *b*).

| *plus*(*a*,*b*) | EBoolean | EInteger | EString |
|---|---|---|---|
| EBoolean | EBoolean | EInteger | EString |
| | *a* **or** *b* | BI(*a*,*b*) | BS(*a*,*b*) |
| EInteger | EInteger | EInteger | EString |
| | IB(*a*,*b*) | *a*+*b* | IS(*a*,*b*) |
| EString | EString | EString | EString |
| | SB(*a*,*b*) | SI(*a*,*b*) | *a*‖*b* |

where the abbreviations are defined as follows

| | | | |
|---|---|---|---|
| BI(*a*,*b*) | if *a* then *b*+1 else *b* | IB(*a*,*b*) | if *b* then *a*+1 else *a* |
| BS(*a*,*b*) | *a.toString*() ‖ *b* | SB(*a*,*b*) | *a* ‖ *b.toString*() |
| IS(*a*,*b*) | *a.toString*() ‖ *b* | SI(*a*,*b*) | *a* ‖ *b.toString*() |
| ‖ | string concatenation | | |

*EStrings* produced by *plus* are restricted to at most 24 characters. (Longer strings are to be truncated silently).

There are several ways to define such a *plus* operator. We'll do so in this assignment by defining separate **static** methods, one for each combination of parameter types, in a separate *Plus* class. Declaring the fields in the element classes as **protected** (as noted in Part 0) allows *Plus* the access it needs since these classes are in the same "program" package. If this were a "real" program, then we would define a separate package for these classes. However, we want to avoid some of the complications involving packages. (These complications are not conceptual, just practical, dealing with classpaths, etc.) (Note that in C++ the *Plus* class would be made a **friend** of the element classes.)

This part is to add the *plus* operators for the cases where both operands are the same type of *Element.*

So, the signatures are

```
public static Element plus (EBoolean, EBoolean)
public static Element plus (EInteger, EInteger)
public static Element plus (EString, EString)
```

Note that the declared type of the object *plus* returns is *Element.* However, in this part, the actual object each *plus* returns is the same type as its arguments.

## Part 4: More Polymorphic Overloaded Plus

This part is to add the operators for the other cases.

Again, note that the declared type of the object *plus* returns is *Element.* However, in this part, the actual object each *plus* returns is as defined in the above table.

## Part 5: Polymorphic Overloaded Plus Revisited

Replace the *plus* methods from the previous two parts with a single *plus* method with signature:

```
public static Element plus (Element, Element)
```

Hint: use **instanceof**, a multi-way if statement, and casting.

## Part 6: Lists of *Elements*

Define an *EList* class, whose items are elements. This class will be similar to one that implements a Java collection, but considerably simpler and with some notable differences. *EList* defines the following methods

```
public void add(Element e)
public int size()
public int capacity()
```

Elements stored in an *EList* are stored in the order in which they are inserted via *add*. *size* returns the number of elements currently in the *EList*.

The implementation of an *EList* must use an array (see "Details") of *Elements* to store its elements. The initial size of this underlying array is 2. When *add* attempts to add an element and the underlying array is full, it allocates a new array of double the current size, copies the existing elements to the new array, and then adds the new element to the new array. Your code must use *System.arraycopy* for this copying. (You could easily write your own copying code, but it's better for you to learn about *System.arraycopy*, which can also be more efficient.) The *capacity* method returns the size of the underlying array. (Thus, the *capacity* method reveals some implementation details outside the class, which in many cases isn't a good idea. We do so for testing purposes.)

(An alternative technique for "growing" the *EList* would, of course, be to use a simple linked list. Since you've probably implemented many linked lists previously and perhaps not used the array based approach, we use the latter for this assignment.)

Override *Elist*'s *toString* method. It is to return a string consisting of each element of the list on a separate line; the list is preceded by a line containing'++++' and followed by a line containing'----'. (Don't add a newline to this last line. See correct output files for exact details.)

## Part 7: Iterators over Lists

Define an *EListIt* class to serve as an iterator (aka generator) over *EList*. It defines the usual iterator methods:

```
// any more elements?
public boolean hasNext()
// return the next element and advance iterator to following item.
public Element next()
```

As usual, *hasNext*() is to have no side effects. If *next*() is invoked when no elements remain, then your program is to print an appropriate error message to *System.err* and then *System.exit(1)*;

Also, add to the *EList* class the method

```
public EListIt iterator()
```

It returns a new *EListIt* for the *EList* object. (You may also need a constructor for the *EListIt* class; details left up to you.) Assume the *EList* is not modified while an *EListIt* is active for it.

Also, add a new class *EListUser*, which contains the method with signature

```
public static int size1(EList k)
```

Its return value is the same a *k.size*(). However, *size1*'s code must use the *EListIt* iterator in a meaningful way. That is, it is to use *hasNext*() and *next*(). It is not to access directly the fields in *EList* (or use access methods in *EList*, etc.); instead it uses *EListIt* to do so indirectly. It may use *ELists*'s *iterator* method. (This should be simple for you.)

Also, add to class *EListUser* the method with signature

```
public static boolean isBalanced(EList k)
```

Its return value indicates whether *Elist k* is balanced in the sense that *k* has the same number of *EInteger* elements in its first half as it does in its second half. Let *N* represents the length of *k*. The first half of *k* consists of its first floor(*N*/2) elements; the second half of *k* consists of its last floor(*N*/2) elements. Note that if *N* is odd, the middle element falls in neither half; it is ignored. Also note that if *k* is empty (*N* is 0), then *k* is balanced.

This problem has a straightforward solution. Your solution is allowed to allocate only a constant amount (i.e., O(1)) of space. Your solution is allowed to use only a linear amount of time (i.e., O(*N*), where *N* is the length of *EList k*).

*isBalanced*'s code must use the *EListIt* iterator in a meaningful way. That is, it is to use *hasNext*() and *next*(). It is not to access directly the fields in *EList* (or use access methods in *EList*, etc.); instead it uses *EListIt* to do so indirectly. It may use *ELists*'s *iterator* method. Your code may use **instanceof.**

## Part 8: Adding Exceptions to Iterators

Change the behavior of *next*() so that if it is invoked when no elements remain, then it just throws a *UsingIteratorPastEndException.* That is, *next*() now has the signature

```
public int next() throws UsingIteratorPastEndException
```

You'll need to define the class *UsingIteratorPastEndException*, with an appropriate constructor (similar to that in the notes; also see the correct output for output details). In practice, you would likely use Java's *NoSuchElementException*, but it's important to see how to declare your own exceptions. Also, Java's *NoSuchElementException* is a *RuntimeException*, which is an *unchecked* exception; recall that means that the exception need not be declared in a method's **throws** clause if the method might throw the exception. In practice, *UsingIteratorPastEndException* should be a *RuntimeException*, but we intentionally don't do that (and instead make it a *checked* exception) just to give you practice with **throw** clauses, etc.

A coding detail: To prevent warnings from *javac* (with *-Xlint*), e.g.,

```
UsingIteratorPastEndException.java:1: warning: [serial] serializable
class UsingIteratorPastEndException has no definition of serialVersionUID
```

put in the top level of the *UsingIteratorPastEndException* class:

```
static final long serialVersionUID = 98L; // any number works here.
```

Because *next*() can now throw an exception, you'll need to revise the body of *size1*() in class *EListUser.* Depending on how you wrote *isBalanced*, you might similarly need to revise its body too. (I didn't need to revise my code.)

Also, add a new method to class *EListUser*:

```
public static int size2(EList k)
```

It computes the same result as *size1*, but in a different way. *size2* can use only the iterator's *next*() method (**not** the *hasNext*() method). It must use **try**/**catch** in meaningful ways. *size2*() must not call (directly or indirectly) *size1* (or *size*). Note: this is not a recommended programming style, but it will help you see how exception handling works. Hint: use **try**/**catch** in place of *hasNext*(); see the test program.

**Part 9: Comparable and Comparators**

(1) We want to be able to compare elements of *EBoolean*, *EInteger*, and *EString* classes with elements of the same class. For example, we want to be able to compare two elements of *EString* (but not to be able to compare an element of *EString* with an element of *EBoolean*). That will allow, for example, lists of such elements to be sorted. The ordering is defined as

- *EBoolean*: false, true. (false is considered less than true.)
- *EInteger*: usual integer ordering.
- *EString*: usual ordering defined by String.

To allow such comparisons, change *EBoolean*, *EInteger*, and *EString* so each implements the *java.lang.Comparable* interface. For example, the declaration of *EBoolean* will now look like

```
public class EBoolean extends Element implements Comparable<EBoolean> {
   ...
   public int compareTo(EBoolean other) {
      ...
   }
}
```

(2) We also want to be able to sort into reverse order lists of *EInteger* objects. To do so, we'll define a new class *EIntegerReverseComparator* that implements the *java.util.Comparator* interface. It looks like:

```
import java.util.Comparator;
public class EIntegerReverseComparator implements Comparator<EInteger> {
   public int compare(EInteger e1, EInteger e2) {
      ...
   }
}
```

(We could also define similar comparators for *EBoolean* and *EString*, but we don't.)

The main program can use this comparator as a parameter to the builtin sort method on lists. For example, if *g1* is an *ArrayList<EInteger>*, we can invoke the builtin sort method via

```
Collections.sort(g1, new EIntegerReverseComparator());
```

*sort*'s second parameter must implement *Comparator*. *sort* invokes the *compare* method in that object. Note how this example demonstrates how in Java to get the effect of "function pointers".

You must write your own *compare* method; do **not** use *Collections.reverseOrder*.

(3) We want to be able to compare all kinds of *Elements*, so that we can, for example, sort a list of *Elements*. We define the ordering among the kinds of *Elements* in increasing order:

- *EBoolean*
- *EInteger*
- *EString*

That means that any *EBoolean* is less than any *EInteger* or *EString*, and that any *EInteger* is less than any *EString*. In comparing *Elements* of the same kind, we use the natural orderings we defined in (1) above. So, define a new Comparator:

```
public class ElementComparator implements Comparator<Element> {
  public int compare(Element e1, Element e2) {
    ...
  }
}
```

Hint: use **instanceof** and casting in *compare*'s code and invoke the methods you wrote for (1).

**Details**

- You must use Java 1.8 (as you did on the previous Java assignment), not Java 1.4.
- You will be given an initial *Element* class and a Makefile for each part. You must use the provided Makefiles. (You won't need to modify them.) It asks the Java compiler to give all possible warnings about your code. Your code for each part must compile without any warnings.
- Each class or interface X must be put in its own X.java file.
- Create **private** methods and fields of your choice, but no **public** or **protected** methods or fields not explicitly specified in this handout.
- Use only **int**s, **boolean**s, **Strings**, and arrays in your code. Do **not** use any predefined Java or other package.

    • For example, do not **use** java.util.ArrayList, anything in java.util.Arrays, java.util.Vector, etc.

    • Exception: you may use methods in String and the println and print methods from System.out and System.err.

    • Exception: you may use Java's *Math.min* and *Math.max.*

    • Exception: the *UsingIteratorPastEndException* class needs to **extend** Java's *Exception* class.

  In particular, using any Java collection or iterator would trivialize parts of this assignment and you will receive no credit. One purpose of this assignment is for you to write your own iterators (some students in previous offerings of this course have never written an iterator class). In particular, using any Java collection (e.g., ArrayList) or iterator would trivialize parts of this assignment and you will receive no credit. One purpose of this assignment is for you to write your own list and iterator classes (some students in previous offerings of this course have never written an iterator class).
- This program has no input data. Instead, you will be given test programs that will employ and exercise your classes. You may modify the tests during debugging (e.g., by commenting out some advanced tests, inserting additional code or tests, etc.) but your program must work on the *unmodified* and, as usual, possibly other tests.
- "Correct" output will also be given. Be sure to use the provided test files and test scripts. Your output must match the "correct" output. By "match", we mean match exactly character by character, including blanks, on each line; also, do not add or omit any blank lines. (Use "make run" or "make runv", which employs diff to test for and display differences.) However, your output might differ in the creation counts (Part 2 and subsequent parts) because their exact values depend on your overall approach. As long as your counts are consistent with your approach, they can differ from the provided correct output (and you don't need to change your code to conform to the provided correct output for this reason). In any case, it is up to you to verify the correctness of your output.

- Express your program neatly. Part of your grade will be based on presentation. Indentation and comments are important. Unless it's immediately obvious, be sure to define each variable used and each method with a concise comment describing its purpose.
- Do not be overly concerned with efficiency. On the other hand, do not write a grossly inefficient program.
- Grading will be divided as follows.

| Percentage | |
|---|---|
| 5 | Part 0: Basic Derivation |
| 5 | Part 1: The hash method |
| 10 | Part 2: Counting Instances |
| 10 | Part 3: Polymorphic Overloaded Plus |
| 10 | Part 4: More Polymorphic Overloaded Plus |
| 10 | Part 5: Polymorphic Overloaded Plus Revisited |
| 10 | Part 6: Lists of *Elements* |
| 15 | Part 7: Iterators over Lists |
| 15 | Part 8: Adding Exceptions to Iterators |
| 10 | Part 9: Comparable and Comparators |

- You must develop your program by parts in the order given. You will be expected to turn in all working parts along with your attempt at the next part. So, be sure to save your work for each part. No credit will be given if the initial working parts are not turned in.
- Points will be deducted for not following instructions, such as those listed herein.
- In seeking assistance on this project, bring your last working part along with your attempt at the next part.
- Work in an incremental style, focusing on a small piece of a part at a time, and getting that to work, before moving on. Unless you are a truly expert programmer and designer, don't try to solve even an entire part at a time.
- A message giving exact details of what to turn in, where the provided source files are, etc. will be posted.