Assignment 3: Java Object-Oriented Programming
Due: Thursday, 12 February 2015 at 11:59pm

Overview

The purpose of this assignment is for you to gain experience with object-oriented programming in Java, and in particular the ideas of inheritance, dynamic binding (virtual methods), overriding and overloading methods. 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, several languages (*e.g.*, Awk, Icon, Lisp, CLUS, SISAL) do provide similar features. Your program needs to implement several abstractions. The first abstraction is an `Element`, of which there are many kinds: `MyInteger`, `MyChar` and `Sequence`. `MyInteger` and `MyChar` respectively are class-based abstractions of `int` and `char`. Class `Sequence` is a dynamic array of `Element` objects. A `Sequence` object, thus stores objects of type `MyInteger`, `MyChar` and `Sequence`. The following is an example of a `Sequence` object:

```
[ 1 'a' [1 3 'b']]
```

In this example, the `Sequence` object contains three `Element` objects: `1`, `'a'` and another `Sequence` object `[1 3 'b']`. Thus, a `Sequence` object may contain nested `Sequence` objects.

Part 1: Class Definitions

The first part of this homework requires you to define and implement the classes and their member methods. Use the names `Element`, `MyInteger`, `MyChar` and `Sequence` for your class definitions. Note that there is an *is-a* relationship between `Element` and `MyInteger`, `MyChar` and `Sequence` classes.

- Define constructor for each class. `MyInteger` objects are initialized to `0`, `MyChar` objects to `'0'`, and `Sequence` to an empty sequence, *i.e.*, a sequence with zero elements.
- Define `Get` and `Set` methods for `Char` and `Integer`:

```
class MyChar extends Element {
  public MyChar() { ... }
  public char Get() { ... }
  public void Set(char val) { ... }
  ...
}

class MyInteger extends Element {
  public MyInteger() { ... }
  public int Get() { ... }
  public void Set(int val) { ... }
  ...
}
```

Methods `Get` and `Set` respectively are used to access and modify the value associated with an object. For instance, invocation of `Get` over a `MyInteger` object returns the integer value associated with the object.

Part 2: Additional Methods

Extend the definitions of the classes by defining the following methods:

1. Define a `Print` member method for each class. Different objects are printed in the following manner:

   - `Element`: Define `Print` to be an abstract method.
   - `MyInteger`: Print the corresponding integer value.
   - `MyChar`: Print the quoted value of a `MyChar` object, *e.g.*, `'c'`.
   - `Sequence`: Print a sequence by surrounding the values of the elements in [,], *e.g.*,

     ```
     [ [ 1 ] [ 2 ] '3' 'c' [ 1 3 ['4' '5'] ] ]
     ```

2. Define the following methods for the `Sequence` class:

   - Define method `first` to return the first element of the sequence:
     `Element first();`
   - Define a method `rest` that returns the rest of the elements of the seqence:
     `Sequence rest();`
     Note that `rest` does not create a new sequence. It merely points to the rest of the elements of the original sequence.
   - Define a method `length` to return the number of elements in a `Sequence` object:
     `int length();`
   - Define a method `add` to add an element at a specified position:
     `void add(Element elm, int pos);`
     If an element already exists at `pos`, `elm` is inserted at `pos`. All elements starting at location `pos` are pushed to the right. If `pos` is not between 0 and the length of the `Sequence` object, flag an error and exit.
   - Define a `delete` method to remove an element at a specified position:
     `void delete(int pos);`
     After deleting the element at `pos`, all elements to the right of `pos` are pushed to the left. If there are no elements at `pos`, the `Sequence` object remains unchanged.

Part 3: Additional Methods of Sequence

Class `Sequence` defines the following additional methods:

1. `index` to access the element at a particular position:
   `public Element index(int pos);`

   The method `index` is used to access a particular element of a `Sequence` object. For instance, `S.index(0)` returns the first element of `S`. If `pos` is not between 0 and the length of the `Sequence` object, flag an error and exit.

2. `flatten` to flatten a sequence:
   `public Sequence flatten();`

   An example usage of `flatten` is shown below:

   ```
   flatten ([1 2 [1 3 4] ['s' a b] ]) = [1 2 1 3 4 's' a b]
   ```

   Note that `flatten` returns a new `Sequence` object.

3. `copy` to perform a *deep copy* of a `Sequence` object:
   `public Sequence copy();`

Part 4: Iterator over Sequence

Define a `SequenceIterator` class to serve as an iterator over `Sequence` objects. An iterator object is used to iterate over a data structure. Note also that there can be multiple `SequenceIterator` objects for a `Sequence` object.

The `Sequence` class must be extended to implement the following member methods:

```
class Sequence extends Element {
  SequenceIterator begin();
  SequenceIterator end();
  ...
}
```

Method `begin` returns a `SequenceIterator` object that points to the first element of the sequence. Method `end`, on the other hand, returns a `SequenceIterator` object that points to a special value after the last element of a `Sequence` object. This special value is used merely to indicate that an iterator has gone beyond the last element of a `Sequence` object. It can be implemented by adding an extra dummy element to a `Sequence` object.

Define the following methods for `SequenceIterator`:

1. `advance` to advance a `SequenceIterator` to the next element in a `Sequence` object.

2. `get` to return the object to which the `SequenceIterator` object points.

3. `equal` to determine if two `SequenceIterator` objects point to the same element.

The interface of the `SequenceIterator` class may look like the following:

```
class SequenceIterator {
  ...
  public bool equal (SequenceIterator other) { ... }
  SequenceIterator advance();
  Element get();
  ...
}
```

The following code fragment shows a possible usage of the iterator:

```
Sequence seq;
SequenceIterator it;

for (it = seq.begin(); it != seq.end(); it.advance()) {
  (it.get()).f();
}
```

The above applies method `f` over all elements of `seq`.

Part 5: Two Dimensional Sequences

Define a class `Matrix` by extending `Sequence`. `Matrix` represents a two dimensional array of integers, and defines the following methods:

```
class Matrix extends Sequence {
  // constructor for creating a matrix of specific number of rows and columns
  Matrix(int rowsize, int colsize);

  void Set(int rowsize, int colsize, int value); // set the value of an element
  int Get(int rowsize, int colsize); // get the value of an element

  Matrix Sum(Matrix mat); // return the sum of two matrices: mat & this
  Matrix Product(Matrix mat); // return the product of two matrices: mat & this
  void Print();  // print the elements of matrix
}
```

Part 6: Map

This part of the homework involves modifying the `Sequence` class so that the elements of a `Sequence` object can be stored and retrieved using a key. We will call this class `Map`. A `Map` object therefore stores a set of `Pair` objects. Each `Pair` object contains a key object and a value object. A key object refers to a `MyChar` object whereas a value object can refer to any type of *Element*. An example of a `Map` object is shown below:

```
[ (‘1’ ‘a’) (‘3’ ‘h’) (‘8’ 4) ]
```

This `Map` object contains three `Pair` objects: `(‘1’ ‘a’)`, `(‘3’ ‘h’)`, and `(‘8’ 4)`. The first component in each `Pair` object is the key object whereas the second is the value object. Note that the elements of the `Map` object are ordered according to the key object. Define the following for the `Map` class:

1. Define an iterator class `MapIterator` for the `Map` class. The behavior of a `MapIterator` object is similar to that of a `SequenceIterator` object. `MapIterator` implements all methods that `SequenceIterator` supports.

2. Methods `begin`, `end`, and `Print`. Methods `begin` and `end` are similar to those of the `Sequence` class (see Part 4). Method `Print` prints a map object in the following manner:

```
[ (‘1’ 1) (‘2’ 2) (‘3’ 8) ]
```

The first is the key object, and the second is the value object.

3. `add` method for adding a `Pair` object:
   `public void add(Pair inval);`

   The method `add` adds a `Pair` object, say `p`, in a `Map` object, say `mp`, by comparing the key value of `p` with the key values of the `Pair` objects of `mp`, and inserts `p` such that the key values of the `Pair` objects are in ascending order. Note that a `Map` object may contain `Pair` objects with identical keys. In the case when there are multiple `Pair` objects with the same key, `p` is added after these `Pair` objects.

4. `find` method for finding a `Pair` object given the key component:
   `public MapIterator find(MyChar key);`

   The argument to the `find` method is the key component of a `Pair` object. It is used to search a `Map` object for a `Pair` object. If the object exists, return a `MapIterator` object that points to the element. If the object does not exist, return a `MapIterator` object that points to the end element. If multiple `Pair` objects with the same key exist, return a `MapIterator` object that points to the first such `Pair` object.

Notes

- **Note that Java provides several of these classes. For this assignment, you can only use pre-existing `Integer`, `Char` and `String` classes for defining `MyInteger`, `MyChar`, and `MyString` respectively. You cannot use any other libraries or pre-existing classes for implementing other classes such as `Sequence` and `Map`. In other words, implement them from scratch.**
- Do not be overly concerned with efficiency. On the other hand, do not write a grossly inefficient program.
- Express your program neatly. Part of your grade will be based on presentation. Indentation and comments are important. Be sure to define each variable used with a concise comment describing its purpose. In general, each procedure should have a comment at its beginning describing its purpose.
- Create several of your own test programs. Your will probably want different test data for the different parts. Be sure that your program works for boundary conditions.
- 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, but you must use the unmodified tests for what you turn in.
- “Correct” output will also be given. Your output can differ in minor ways yet still be correct. It is up to you to verify your code’s correctness. In general, the easiest thing to do is to make your output conform to the given correct output and then use “make run”, which employs diff to test for and display differences.
- You must develop your program in parts as outlined above. Grading will be divided as follows:

Part	Percentage
1	10
2	20
3	15
4	20
5	15
6	20

If your program does not fully work, hand in a listing of the last working part along with your attempt at the next part, and indicate clearly which is which. No credit will be given if the last working part is not turned in. Points will be deducted for not following instructions, such as the above.

- Work in 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, do not try to solve the whole program, or even an entire part, at a time.
- A message giving exact details of what to turn in, where the provided test files are, etc, will be posted on the class web site.
- **Get started now to avoid the last minute rush.**