Programming Project 2
CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development
March 30, 2016

Last revision March 30, 2016

0. Introduction.

In this assignment, you must write a Java class that takes a *partial ordering*, and constructs a sequence of objects that satisfies the ordering. The theory section explains what a partial ordering is, and how a sequence can satisfy it. The implementation section explains the algorithm that you must use, and some advice about how to write it in Java.

1. Theory.

Suppose that a and b are integers. Also suppose that $a \neq b$, and that $<$ is the usual less-than relation. Then either $a < b$, or $b < a$. Since this is true for all choices of a and b, we say that the set of integers is *totally ordered* under $<$.

Suppose instead that a and b are objects, not necessarily integers. Also suppose that $a \neq b$, and that $<$ is a relation on those objects. For *some* choices of a and b, either $a < b$, or $b < a$. Since this is not true for all choices of a and b, we say that the set of these objects is *partially ordered* under $<$.

A partial ordering like $<$ can model how a series of steps is to be performed over time, if $a < b$ means that step a must be performed before step b. For example, a college student who is majoring in Mathematics might be required to take the courses MATH 101, MATH 102, MATH 103, and MATH 104, in that order. If the student is also minoring in Art, then he or she might be required to take the courses ART 101, ART 102, and ART 103, in that order. These requirements can be modeled by the following partial ordering.

MATH 101 < MATH 102 ART 101 < ART 102
MATH 102 < MATH 103 ART 102 < ART 103
MATH 103 < MATH 104

Note that $<$ does not say whether Mathematics courses must be taken before Art courses, whether Art courses must be taken before Mathematics courses, or whether courses in both subjects can be taken at the same time.

Now suppose that there is a linear sequence of objects, and that those objects are partially ordered by $<$. Also suppose that if $a < b$, then a appears before b in the sequence. Then the sequence is said to *satisfy* the partial ordering. For example, taking Art and Mathematics courses in any of these sequences will satisfy the partial ordering discussed previously.

MATH 101 · MATH 102 · MATH 103 · MATH 104 · ART 101 · ART 102 · ART 103

ART 101 · ART 102 · ART 103 · MATH 101 · MATH 102 · MATH 103 · MATH 104

MATH 101 · ART 101 · MATH 102 · ART 102 · MATH 103 · ART 103 · MATH 104

ART 101 · MATH 101 · MATH 102 · ART 102 · MATH 103 · MATH 104 · ART 103

These are not the only sequences that satisfy $<$. In general, a partial ordering may be satisfied by many different sequences.

Some partial orderings cannot be satisfied by *any* linear sequence of objects. One such ordering is $a < b$

and $b < a$. Orderings like this can occur in real life. For example, there are said to be professions in which you cannot get a job until you join a union, but you cannot join a union until you get a job.

Finally, suppose that there is a set of objects which is partially ordered by <. Then there is an algorithm that can construct a sequence of these objects, which satisfies the ordering. The algorithm starts with an empty sequence. It first removes from the set a *minimal* object that has no predecessor according to <. Then it adds that object to the end of the sequence. It continues in this way, until the set becomes empty, or until no more objects can be removed. If the set is empty, then the sequence satisfies <. If no more objects can be removed, then no sequence satisfies <.

Other algorithms for satisfying a partial ordering are more efficient than the one you will use for this project. However, this one is easy to implement. It also involves manipulating linear, singly-linked lists in various ways, which is the real point of it all.

2. Implementation.

You must write a Java class called `Ordering` that looks like this. The three dots stand for the Java code that you must write. To make this assignment easier to grade, you must use the same names for things as are used here.

```
class Ordering<Base>
{
  ⋮
}
```

The class `Ordering` must have two nested classes, called `Pair` and `Element`. The class `Pair` must have three slots: a `Base` slot called `left`, another `Base` slot called `right`, and a `Pair` slot called `next`. The class `Element` must have two slots: a `Base` slot called `object`, and an `Element` slot called `next`.

The class `Ordering` must also have two private variables, called `pairs` and `elements`. The variable `pairs` must be a linear, singly-linked list, made by linking instances of `Pair` through their `next` slots. The list `pairs` represents a partial ordering. If $a < b$ in the partial ordering, then there is a `Pair` in the list whose `left` slot is a, and whose `right` slot is b.

The variable `elements` must be a linear, singly-linked list, made by linking instances of `Element` through their `next` slots. The `object` slots in the list make up the set of objects to be organized into a sequence that satisfies the partial ordering in `pairs`.

Finally, the class `Ordering` must have the following methods. They must work as described here. To make your code easier to write, many of these methods are designed to call other methods.

```
public Ordering()
```

> Constructor. Make a new instance of the class `Ordering`. Initialize the lists `elements` and `pairs` so they are empty. The list `elements` must have a head node, but the list `pairs` must not.

```
private boolean isEmpty()
```

> Test if `elements` is empty.

```
private boolean isElement(Base object)
```

> Test if `object` appears in `elements`.

```
private boolean isPair(Base left, Base right)
```

Test if there is a `Pair` in `pairs` whose `left` slot is the parameter `left`, and whose `right` slot is the parameter `right`.

```
private boolean isMinimum(Base right)
```

Visit each object `left` in `elements`. Test if there is no `Pair` in `pairs` with the object `left` in its `left` slot, and the object `right` in its `right` slot. In other words, test if `right` is not preceded by any other object in `elements`, according to the partial ordering in `pairs`. Hint: use `isPair`.

```
private Base minimum()
```

Find a minimum object in `elements`, one that is not preceded by another object in `elements`, according to the partial ordering in `pairs`. Delete the minimum object from `elements`, and return that object. Throw an `IllegalStateException` if there is no minimum object in `elements`. Hint: use `isMinimum`.

```
public void precedes(Base left, Base right)
```

If `left` or `right` is `null`, then throw an `IllegalArgumentException`. Otherwise, add a new `Pair` to `pairs` whose `left` slot is the parameter `left`, and whose `right` slot is the parameter `right`. Also, if `left` is not in `elements`, then add it, and if `right` is not in `elements`, then add it as well. This method says that `left` precedes `right` in a partial ordering. It also says what objects are to be in the sequence that will satisfy the partial ordering. Hint: use `isElement`.

```
public String satisfy()
```

Use the algorithm of the previous section to find a sequence of objects in `elements` that satisfies the partial ordering in `pairs`. Make a `String` of these objects, separated by '<'s, and return that `String` (see the example below). Hints: use `minimum`, and the `toString` methods of the objects. Also consider using the `StringBuffer` or `StringBuilder` classes.

All these methods require only a few lines of code each. If you find yourself writing many lines of code for any of these methods, then you are probably making a mistake.

3. Example.

You must also write a driver class that makes an instance of your `Ordering` class and tests its methods. Here's what it might look like.

```
class PastramiOnRye
{
  public static void main(String [] args)
  {
    Ordering<String> ordering = new Ordering<String>();

    ordering.precedes("A", "B");
    ordering.precedes("A", "C");
    ordering.precedes("B", "D");
    ordering.precedes("B", "J");
    ordering.precedes("C", "E");
```

```
        ordering.precedes("D", "F");
        ordering.precedes("D", "H");
        ordering.precedes("E", "H");
        ordering.precedes("F", "C");
        ordering.precedes("G", "E");
        ordering.precedes("G", "I");
        ordering.precedes("I", "D");
        ordering.precedes("I", "J");

    //  This should print G < I < A < B < J < D < F < C < E < H.

        System.out.println(ordering.satisfy());
      }
    }
```

This is not necessarily a complete test of the `Ordering` class. You may want to include some more tests of your own design. Also, you may print an answer that is different from this one, but is still correct.

4. Deliverables.

Unlike the lab assignments, you are not allowed to work with a partner on this project. IT MUST BE COMPLETED BY YOURSELF ALONE. Although you can discuss the assignment in a general way with others, you are not allowed to get help from anyone, except Prof. Moen or the course TA's.

The project is worth 60 points. It is due in two weeks, at midnight on April 13, 2016. At that time, you must turn in Java code for your `Ordering` class. It must include the nested classes `Element` (5 points) and `Pair` (5 points.) It must also include an `Ordering` constructor (5 points), and the methods `isEmpty` (5 points), `isElement` (5 points), `isMinimum` (5 points), `isPair` (5 points), `minimum` (10 points), `precedes` (5 points), and `satisfy` (10 points). You must also turn in Java code for your driver class, and any output it produces, even though you don't get points for them. Put the code for both classes in one file, and put the output in a comment at the end of the file. If you have questions about how to turn in your project, then ask your lab TA.