Subclassing for ADTs Implementation

An interesting use of subclassing is to implement some forms of ADTs more cleanly, especially ADTs that have different representations for their values.

Let me use a simple ADT that you are familiar with, integer lists, with the following signature and specification:

```
public static List empty ();
public static List cons (int, List);
public boolean isEmpty ();
public int first ();
public List rest ();
public String toString ();

empty().isEmpty() = true
cons(i,s).isEmpty() = false
cons(i,s).first() = i
cons(i,s).rest() = s
empty().toString() = "<empty>"
cons(i,s).toString() = i + " " + s.toString()
```

Consider the following direct implementation of the List ADT as a linked list:

```
public class List {
  private Integer firstElement;
  private List restElements;

  private List (Integer i, List s) {
    firstElement = i;
    restElements = s;
  }

  public static List empty () {
    return new List(null,null);
  }
```

```
  public static List cons (int i, List s) {
    return new List(new Integer (i),s);
    // class Integer lets you use null as a value
  }

  public boolean isEmpty () {
    return (this.firstElement == null);
  }

  public int first () {
    if (this.isEmpty())
      throw new IllegalArgumentException("first() on empty list");
    return this.firstElement;
  }

  public List rest () {
    if (this.isEmpty())
      throw new IllegalArgumentException("rest() on empty list");
    return this.restElements;
  }

  public String toString () {
    if (this.isEmpty())
      return "<empty>";
    return this.first() + " " + this.rest();
  }
}
```

This code and its underlying representation is ugly, and I mean that as a technical term. Part of the problem is that there is an implicit invariant: whenever the `firstElement` field is not null, then the `restElements` field better not be null either. (Can you see what can go wrong if we do not respect this invariant?) We must make sure that the implementation always preserves that invariant. There are ways around that, making the invariant more robust, but they're a bit unsatisfying.

Another problem with the implementation is that there are all kinds of checks in the code that test whether the list is empty or not. It would be much better to instead have two kinds of lists, an empty list and a "cons list", and have them implement their respective methods knowing full well what their representation is. This is what we're going to explore now. We will use subclasses to keep track of the kind of list we have, and the subclasses will implement their methods in full confidence that the list they are manipulating is of the right kind, without needing to check the representation.

But that means that the `List` class, by itself, does not represent anything. The representation is in the subclasses. It does not make sense to create an object of type `List` anymore. We will only create an object of class `EmptyList` or `ConsList`, as they will be named. To enforce this, we are going to make the `List` class `abstract`. An abstract class is a class that cannot be instantiated.[1] Therefore, it does not have a Java constructor. The only use of an abstract class is to serve as a superclass for other classes. An abstract class need not implement all its methods. It can have abstract methods that simply promise that subclasses will implement those methods. (Java will enforce this, by making that all concrete subclasses of the abstract class do provide an implementation for the methods.)

Here is a recipe for implementing an immutable ADT that is specified by an algebraic specification using subclasses.

Let T be the name of the ADT.

Assumptions on the structure of the ADT interface:

- Assume that the signature is given in OO-style: except for the basic creators, each operation of the ADT takes an implicit argument which is an object of type T.
- Except for the basic creators, each operation is specified by one or more equations. If an operation is specified by more than one equation, then the left hand sides of the equations differ according to which basic creator was used to create an argument of type T.
- The equations have certain other technical properties that allow them to be used as rewriting rules.
- We are to implement the ADT in Java.
- The creators of the ADT are to be implemented as static methods of a class named T.
- Other operations of the ADT are to be implemented as methods of a class named T.

Steps of the recipe:

- Determine which operations of the ADT are basic creators and which are other operations.
- Define an abstract class named T.
- For each basic creator c, define a concrete subclass of T whose instance variables correspond to the arguments that are passed to c. For each such subclass, define a Java constructor that takes the same arguments as c and stores them into the instance variables.

[1] In contrast, a class that *can* be instantiated is sometimes called a *concrete* class.

(So far we have defined the representation of T. Now we have to define the operations.)

- For each creator of the ADT, define a static method within the abstract class that creates and returns a new instance of the subclass that corresponds to c.
- For each operation f that is not a basic creator, define an abstract method f.
- For each operation f that is not a basic creator, and for each concrete subclass C of T, define f as a dynamic method within C that takes the arguments that were declared for the abstract method f and returns the value specified by the algebraic specification for the case in which Java's special variable `this` will be an instance of C. If the algebraic specification does not specify this case, then the code for f should throw a `RuntimeException` such as an `IllegalArgumentException`.

Following this recipe for the List ADT above gives the following. Can you match the steps with what gets produced?

```
public abstract class List {

  public static List empty () {
    return new EmptyList ();
  }

  public static List cons (int i, List s) {
    return new ConsList (i,s);
  }

  public abstract boolean isEmpty ();
  public abstract int first ();
  public abstract List rest ();
  public abstract String toString ();

}



class EmptyList extends List {
  public EmptyList () { }

  public boolean isEmpty () { return true; }

  public int first () {
    throw new IllegalArgumentException ("first() on empty list");
```

```
  }

  public List rest () {
    throw new IllegalArgumentException ("rest() on empty list");
  }

  public String toString () {
    return "<empty>";
  }
}



class ConsList extends List {
  private int firstElement;
  private List restElements;

  public ConsList (int i, List s) {
    firstElement = i;
    restElements = s;
  }

  public boolean isEmpty () { return false; }

  public int first () { return this.firstElement; }

  public List rest () { return this.restElements; }

  public String toString () { return this.first() + " " + this.rest(); }
}
```

Nested Classes

Last time, we saw a recipe for deriving an implementation from an ADT specification. There are two problems with that approach, however, one minor, one major:

(1) Namespace pollution

(2) Extensibility

Let's take care of the minor one now. We'll return to the extensibility problem later in the course.

Consider the `List` class as derived from the recipe. It consists of a public abstract class `List`, and two subclasses, `EmptyList` and `ConsList`. Now, because of the way Java works, either all of those classes must be made public, or, in the case where you decide to put all the classes in the same file, only `List` is made public, and the two subclasses are unannotated. This makes the two subclasses *package-public*: they are visible to other classes in the same package, but unavailable to classes outside the package. In other word, they are public for classes within the same package, but private for classes outside the package.

In practice, the only class we really want to be able to create instances of `EmptyList` and `ConsList` is the `List` class. All list object creation should go through the `List` class static methods. In parts, this is because we rarely want to have objects specifically of class `EmptyList` or `ConsList` about, but rather only want to have `List` objects. The static creators in the `List` class ensure this is the case.

So how do we prevent `EmptyList` and `ConsList` from being available even from within the package. The answer, which may or may not be obvious, is to simply package up the two subclasses directly within the `List` class, and make them private so that they cannot be accessed from outside the class. Here is the code:

```
public abstract class List {

  public static List empty () {
    return new EmptyList ();
  }

  public static List cons (int i, List s) {
    return new ConsList (i,s);
  }

  public abstract boolean isEmpty ();
  public abstract int first ();
  public abstract List rest ();
  public abstract String toString ();


  private static class EmptyList extends List {
    public EmptyList () { }

    public boolean isEmpty () { return true; }

    public int first () {
```

```
      throw new IllegalArgumentException("first() on empty list");
    }

    public List rest () {
      throw new IllegalArgumentException("rest() on empty list");
    }

    public String toString () { return "<empty>"; }
  }


  private static class ConsList extends List {
    private int firstElement;
    private List restElements;

    public ConsList (int i, List s) {
      firstElement = i;
      restElements = s;
    }

    public boolean isEmpty () { return false; }

    public int first () { return this.firstElement; }

    public List rest () { return this.restElements; }

    public String toString () { return this.first() + " " + this.rest(); }
  }
}
```

These are examples of *nested classes*. In fact, these are *static* nested classes. A static nested class is just like a class defined in its own file, except that it is defined within another class. (If a nested class T is defined as public within a class S, then class T can be accessed from outside S as $S.T$.)

Why do we need the static qualifier? The static qualifier, as usual, indicates that the nested class is associated with the class definition itself. In particular, the nested class cannot access instance variables of the class containing it. (Nor can it invoke instance methods of the class containing it.) This essentially says that the nested class is defined only once, when the containing class is itself defined. Note that this is a "containment" relation. Class S contains class definition T, just like it contains static methods and static fields.

(Without the static qualifier, a nested class (say T) is associated with instances of the class

containing the definition (say S). Every instance of S "redefines" the nested class T. In particular, the nested class can refer to instance variables of S. When they are not static, nested classes are called *inner classes*. Inner classes are very powerful, and are related to closures, which can be used to do higher-order programming. In other words, inner classes give you `lambda`. We will not use inner classes much in this course, but it's good to know that they exist.)

Nesting classes takes care of the namespace pollution problem. As I said, we will return to the extensibility problem later in a few lectures.