Object-Oriented Design | **Lecture 11**
CS 3500 Spring 2010 (Pucella) | **Tuesday, Feb 16, 2010**

11 Polymorphism

The functional iterator interface we have defined last lecture is nice, but it is not very general. As defined, it can only be used to iterate over structures that yield integers. (Because of the definition of the `element()` method. If we wanted to define an iterator over structures that yields, say, Booleans, or `Person` objects, then we need to define a new iterator interface for that type. That's suboptimal, to say the least. In particular, this means that we cannot write a function that can use any iterator independently of the type of value it yields. For instance, consider the following function that uses an iterator to count how many elements are in an aggregate structure:

```
public static int countElements (FuncIteratorInt it) {
  if (it.hasElement())
    return 1 + countElements(it.moveToNext());
  else
    return 0;
}
```

That works for iterators that return integers. If you wanted to count the number of lines in a sprite (cf., Homework 2), then you would need to write:

```
public static int countElements (FuncIteratorLine it) {
  if (it.hasElement())
    return 1 + countElements(it.moveToNext());
  else
    return 0;
}
```

That's silly — we need two functions to count elements, and the body of the functions are actually exactly the same, because counting elements does not depend on the type of elements. Can we figure out a nice way to reuse the code for `countElements()` across two different kind of iterators?

11.1 Polymorphic Interfaces

One way to do that is to recognize that the interfaces `FuncIteratorInt` and `FuncIteratorLine` are really the same interface that differs only in the type of the elements. What we really want is an interface that is *parameterized* by the type of result it returns.

```
public interface FuncIterator<T> {
  public boolean hasElement ();
  public T element ();
  public FuncIterator<T> moveToNext ();
}
```

Basically, the definition is as before, except for the `<T>` annotation. This defines interface `FuncIterator` with a type parameter `T`. (In Java, this is called a *generic interface*, but common names are *parameterized* interface, or *polymorphic* interface. I tend to favor the latter.) Within `FuncIterator<T>` you can use `T` as if it were a type. Note the definition/use distinction: the `T` in `interface FuncIterator<T>` is a *definition* — it tells you that `T` is a parameter that can be instantiated (a bit like the parameter of a function) — while every other use of `T` in the interface is a *use* — it tells you that whatever you choose to instantiate `T` with will be used to replace those `T`s.

When it actually comes time to use such an interface, you must *instantiate* it at the type you require, say `FuncIterator<Integer>` or `FuncIterator<Person>`. Intuitively, `FuncIterator<Integer>` is as if you had written `FuncIterator` with `Integer` in place of every `T`. One restriction we have is that you can only instantiate a parameter at a class type — meaning that you cannot write `FuncIterator<int>`, for example. But we can use the classes corresponding to primitive types, `Integer`, `Boolean`, and so on, that are simple wrappers around the primitive types.[1]

Before we get to write a single version of `countElements()` that works for any kind of `FuncIterator`, let me give you an example of how you can define a class that implement such a parameterized interface. Suppose we wanted to have the `List` class use the above interface for its iterator. Recall that `List` is an implementation of integer lists. Here is the resulting implementation:

```
/* ABSTRACT CLASS FOR LISTS */
public abstract class List {

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

  public static List cons (int i, List l) {
    return new ConsList(i,l);
```

[1]Java can and will automatically convert between wrapper classes and primitive types, but sometimes we will create such values by hand, using for instance `new Integer(i)` to create a wrapped integer `i` of type `Integer`, and using method `intValue()` to get the underlying primtive integer out of an object of class `Integer`.

```
  }

  public abstract boolean isEmpty ();

  public abstract int first ();

  public abstract List rest ();

  public abstract FuncIterator<Integer> funcIterator ();
}


/* CONCRETE CLASS FOR EMPTY CREATOR */
class EmptyList extends List {

 public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }

  public int first () {
    throw new Error ("EmptyList.first()");
  }

  public List rest () {
    throw new Error ("EmptyList.rest()");
  }

  public FuncIterator<Integer> funcIterator () {
    return new EmptyFuncIterator();
  }
}


/* ITERATOR FOR EMPTY LISTS */
class EmptyFuncIterator implements FuncIterator<Integer> {

  public EmptyFuncIterator () {}

  public boolean hasElement () {
    return false;
  }

  public Integer element () {
   throw new java.util.NoSuchElementException("EmptyFuncIterator.element()");
  }

  public FuncIterator<Integer> moveToNext () {
   throw new java.util.NoSuchElementException("EmptyFuncIterator.moveToNext()");
  }
}


/* CONCRETE CLASS FOR CONS CREATOR */
class ConsList extends List {

  private int firstElement;
  private List restElements;

  public ConsList (int f, List r) {
    firstElement = f;
    restElements = r;
  }

  public boolean isEmpty () {
    return false;
  }

  public int first () {
    return firstElement;
  }

  public List rest () {
    return restElements;
  }

  public FuncIterator<Integer> funcIterator () {
    return new ConsFuncIterator(firstElement,
                                restElements.funcIterator());
  }
}
```

```
/* ITERATOR FOR NON−EMPTY LISTS */
class ConsFuncIterator implements FuncIterator<Integer> {

  private int currentElement;
  private FuncIterator<Integer> restIterator;

  public ConsFuncIterator (int c, FuncIterator<Integer> r) {
    currentElement = c;
    restIterator = r;
  }

  public boolean hasElement () {
    return true;
  }

  public Integer element () {
      return new Integer(this.currentElement);
  }

  public FuncIterator<Integer> moveToNext () {
    return restIterator;
 }
}
```

In other words, nothing special, aside from the explicit conversion from `int` to `Integer` in method `element()` of class `ConsFuncIterator`, which is not necessary because Java will perform the conversion automatically, but I'm emphasizing here because it will become useful later in the lecture.

To use such a parameterized `FuncIterator` interface, we need to specify exactly how to instantiate the `T` parameter in the definition. Thus, for instance, we can write the following function that prints all the elements that are supplied by an integer-yielding iterator (written using a while loop):

```
  public static void printElements (FuncIterator<Integer> it) {
    FuncIterator<Integer> temp = it;

    while (temp.hasElement()) {
      System.out.println ("Element = " + temp.element());
      temp = temp.moveToNext();
    }
  }
```

or the function that counts the number of elements supplied by an integer-yielding iterator (written recursively):

```
public static int countElements (FuncIterator<Integer> it) {
  if (it.hasElement())
    return 1 + countElements(it.moveToNext());
  else
    return 0;
}
```

Polymorphic interfaces are extensively used in the Java Collections framework.

11.2 Polymorphic Methods

Adding parameterization to interfaces is such a natural addition to a language that it hardly seems worth making a big fuss about it. However, from this small addition, a cascade of other changes naturally follow that drastically affect the programming experience.

Parameterization for interfaces is a way to reuse code—it kept us from having to define multiple interfaces that look the same except for the type of some of their operations. But to maximize code reuse in the presence of polymorphic interfaces, however, we need a bit more than what we have seen until now.

Consider the example I gave at the beginning of the lecture, defining a `countElements()` function that works irrespectively of the kind of functional iterator we have. What is the type of such a function? Intuitively, we would like to say that `countElements()` takes an argument of type `FuncIterator<T>` for *any class T* we want. In pseudo-Java, we would express this as:

```
public static for any T int countElements (FuncIterator<T> it) {
  if (it.hasElement())
    return 1 + countElements(it.moveToNext());
  else
    return 0;
}
```

In real Java, this is written:

```
public static <T> int countElements (FuncIterator<T> it) {
  if (it.hasElement())
    return 1 + countElements(it.moveToNext());
  else
    return 0;
}
```

The signature has that extra `<T>` at the front, before the return type. This is Java-speak for *for any T*, it's the indication that the method is polymorphic, and the `T` in the angle brackets tells you what is the *type variable* that you are using in the method definition. The `<T>` here is the *definition* — every other occurrence of `T` in the function is a *use*. The `T` can be used in the type of the result and the arguments to the method, as well as in the body of the method, in case we need to define local variables that depend on that type.

For comparison, here is a variant of `countElements()` that uses a while loop instead — note how the use of `T` in the definition of the local variable `temp`:

```
public static <T> int countElements (FuncIterator<T> it) {
  FuncIterator<T> temp = it;
  int count = 0;

  while (temp.hasElement()) {
    count = count + 1;
    temp = temp.moveToNext();
  }
  return count;
}
```

When a polymorphic function is called, the type checker tries to figure out what is the right type to instantiate the type variable to. For instance, if you call `countElements(v)` where `v` is a variable with compile-time type `FuncIterator<Line>`, then the type checker figures out that it should call `countElements()` instantiating the type variable `T` to `Line`, and type check the code accordingly.

A polymorphic function doesn't need to use polymorphic interfaces. Consider the problem of writing a single identity function that works for any class. Intuitively, the identity function takes an argument `x` and just returns `x`, without doing anything with it. Such a function should be able to take an `x` of any type `T`, and return that `x`, thus with result type the same `T`. We can write this using a single polymorphic function:

```
public static <T> T identity (T val) {
  return x;
}
```

As an exercise, write a single `printElements()` function that prints all the elements from an iterator, irrespectively of the kind of iterator is given as an argument.

The above examples illustrate that *polymorphic methods are a way to reuse client code,* by only requiring you to write a single client method that works with multiple (possibly unrelated) classes, including those implementing polymorphic interfaces.

Bounded Polymorphism

Let's look at a slightly more complex example. Suppose we write an function that sums all the elements given by a functional iterator, something like:

```
public static int sumElements (FuncIterator<Integer> it) {
  if (it.hasElement())
    return it.element() + sumElements(it.moveToNext());
  else
    return 0;
}
```

(Note that there are implicit conversions between `int`s and `Integer`s in this code.) Clearly, this functions cannot work for all iterators, because not all classes that an iterator can yield have a notion of "addition" defined on it — it doesn't always make sense to add all elements that an iterator yields.

But it should work for *all iterators that yield values for a type that has a notion of addition.* So can we get that kind of code reuse? Yes, using something called *bounded polymorphism.* Basically, bounded polymorphism lets us say "for all types *that satisfy a certain property*," where that property is expressed using subclassing.

So let's first try to figure out how to express the property "has a notion of addition." One way to do that is to define a class with an operation `add`, and since any subclass of that class will be required to have such an operation, that can be taken to mean that it supports addition. We'll make that class an interface. (Why?)

```
public interface Addable<T> {
  T add (T val);
}
```

A class subclassing `Addable<T>` says that it can add an element of type `T` to get element of type `T`. Now, `Integer`s do not implement `Addable`, but we can define a notion of addable integers easily enough:

```
public class AInteger implements Addable<AInteger> {

  private int intValue;

  private AInteger (int i) {
    intValue = i;
  }

  public static AInteger create (int i) {
    return new AInteger(i);
  }

  public int intValue () {
    return this.intValue;
  }

  public AInteger add (AInteger val) {
    return new AInteger(this.intValue()+val.intValue());
  }

  public String toString () {
    return (new Integer(this.intValue())).toString();
  }
}
```

(I should define `equals()` and `hashCode()` methods as well, but I leave them as an exercise.)

Let's modify our `List` implementation so that it uses `AInteger`s instead of `Integer`s:

```
/* ABSTRACT CLASS FOR LISTS */
public abstract class List {

  // no java constructor for this class, it is abstract

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

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

  public abstract boolean isEmpty ();

  public abstract int first ();

  public abstract List rest ();

  public abstract FuncIterator<AInteger> funcIterator ();
}
```

```
/* CONCRETE CLASS FOR EMPTY CREATOR */
class EmptyList extends List {

 public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }

  public int first () {
    throw new Error ("first() on an empty list");
  }

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

  public FuncIterator<AInteger> funcIterator () {
    return new EmptyFuncIterator();
  }
}


/* ITERATOR FOR EMPTY LISTS */
class EmptyFuncIterator implements FuncIterator<AInteger> {

  public EmptyFuncIterator () {}

  public boolean hasElement () {
    return false;
  }

  public AInteger element () {
   throw new java.util.NoSuchElementException("EmptyFuncIterator.element()");
  }

  public FuncIterator<AInteger> moveToNext () {
   throw new java.util.NoSuchElementException("EmptyFuncIterator.moveToNext()");
  }
}
```

```
/* CONCRETE CLASS FOR CONS CREATOR */
class ConsList extends List {

  private int firstElement;
  private List restElements;

  public ConsList (int f, List r) {
    firstElement = f;
    restElements = r;
  }

  public boolean isEmpty () {
    return false;
  }

  public int first () {
      return firstElement;
  }

  public List rest () {
    return restElements;
  }

  public FuncIterator<AInteger> funcIterator () {
    return new ConsFuncIterator(firstElement,
                                restElements.funcIterator());
  }
}


/* ITERATOR FOR NON-EMPTY LISTS */
class ConsFuncIterator implements FuncIterator<AInteger> {

  private int currentElement;
  private FuncIterator<AInteger> restIterator;

  public ConsFuncIterator (int c, FuncIterator<AInteger> r) {
    currentElement = c;
    restIterator = r;
  }

  public boolean hasElement () {
    return true;
  }

  public AInteger element () {
     return AInteger.create(currentElement);
  }

  public FuncIterator<AInteger> moveToNext () {
    return restIterator;
 }
}
```

I can write a *bounded polymorphic* function `sumElements` that sums all the elements given by an iterator as long as the type of values that the iterator yields is a subclass of `Addable`, e.g., has an `add()` operation:

```
  public static <T extends Addable<T>> T sumElements (T init, FuncIterator<T> it) {
    if (it.hasElement())
      return it.element().add(sumElements(init,it.moveToNext()));
    else
      return init;
  }
```

Note that parameterization `<T extends Addable<T>>`, read "for all classes `T` that are subclasses of `Addable<T>`",[2] that is: for all classes `T` that have an operation `add` that take elements of type `T` (same type!) yielding elements of type `T`, which is exactly what we want. *Question: why do we need to supply an initial value of type `T`?*

Now we can write, for example:

```
  List sample = List.cons(1, List.cons(2, List.cons(3, List.empty())));

  AInteger zero = AInteger.create(0);
  AInteger sum = sumElements(zero, sample.funcIterator());
  System.out.println("Sum = " + sum);
```

and get

```
  Sum = 6
```

[2]`extends` in this particular context really means "is a subclass of" — it matches both a class that extends another class, and a class that implements an interface.

as output.

To illustrate the usefulness of this, it would be nice to have another subclass of `Addable<T>`, so we can reuse the above code. Let's define pairs of integers, with an `add` operation that is just vector addition: $(a, b) + (c, d)$ is just $(a + c, b + d)$:

```
public class PairAI implements Addable<PairAI>
{

  private AInteger first;
  private AInteger second;

  private PairAI (AInteger f, AInteger s) {
    this.first = f;
    this.second = s;
  }

  public static PairAI create (AInteger f, AInteger s) {
    return new PairAI(f,s);
  }

  public AInteger first () {
      return this.first;
  }

  public AInteger second () {
      return this.second;
  }

  public String toString () {
      return "(" + this.first().toString() + "," + this.second().toString() + ")";
  }

  public PairAI add (PairAI p) {
    return create(this.first().add(p.first()), this.second().add(p.second()));
  }
}
```

Let's define an ADT for lists of pairs of integers, `ListPairAI`. The signature is the expected one:

```
public static ListPairAI empty ();
public static ListPairAI cons (PairAI, ListPairAI);
```

```
  public boolean isEmpty ();
  public PairAI first ();
  public ListPairAI rest ();
  public String toString ();
  public FuncIterator<PairAI> funcIterator ();
```

The implementation is obtained readily from the Specification design pattern, and some experience writing functional iterators for lists:

```
/* ABSTRACT CLASS FOR LISTS */
public abstract class ListPairAI {

  public static ListPairAI empty () {
    return new EmptyListPairAI();
  }

  public static ListPairAI cons (PairAI p, ListPairAI l) {
    return new ConsListPairAI(p,l);
  }

  public abstract boolean isEmpty ();

  public abstract PairAI first ();

  public abstract ListPairAI rest ();

  public abstract FuncIterator<PairAI> funcIterator ();
}


/* CONCRETE CLASS FOR EMPTY CREATOR */
class EmptyListPairAI extends ListPairAI {

 public EmptyListPairAI () {}

  public boolean isEmpty () {
    return true;
  }

  public PairAI first () {
    throw new Error ("EmptyListPairAI.first()");
  }
  public ListPairAI rest () {
    throw new Error ("EmptyListPairAI.rest()");
  }

  public FuncIterator<PairAI> funcIterator () {
    return new EmptyPAIFuncIterator();
  }
}


/* ITERATOR FOR EMPTY LISTS */
class EmptyPAIFuncIterator implements FuncIterator<PairAI> {

  public EmptyPAIFuncIterator () {}

  public boolean hasElement () {
    return false;
  }

  public PairAI element () {
   throw new java.util.NoSuchElementException("EmptyPAIFuncIterator.element()");
  }

  public FuncIterator<PairAI> moveToNext () {
   throw
     new java.util.NoSuchElementException("EmptyPAIFuncIterator.moveToNext()");
  }
}


/* CONCRETE CLASS FOR CONS CREATOR */
class ConsListPairAI extends ListPairAI {

  private PairAI firstElement;
  private ListPairAI restElements;

  public ConsListPairAI (PairAI f, ListPairAI r) {
    firstElement = f;
    restElements = r;
  }
  public boolean isEmpty () {
    return false;
  }

  public PairAI first () {
    return firstElement;
  }

  public ListPairAI rest () {
    return restElements;
  }

  public FuncIterator<PairAI> funcIterator () {
    return new ConsPAIFuncIterator(firstElement,
                                   restElements.funcIterator());
  }
}


/* ITERATOR FOR NON-EMPTY LISTS */
class ConsPAIFuncIterator implements FuncIterator<PairAI> {

  private PairAI currentElement;
  private FuncIterator<PairAI> restIterator;

  public ConsPAIFuncIterator (PairAI c, FuncIterator<PairAI> r) {
    currentElement = c;
    restIterator = r;
  }

  public boolean hasElement () {
    return true;
  }

  public PairAI element () {
      return currentElement;
  }

  public FuncIterator<PairAI> moveToNext () {
    return restIterator;
 }
}
```

We can check that we can reuse both `printElements()` and `sumElements()`, as expected:

```
  PairAI p1 = PairAI.create(AInteger.create(1),AInteger.create(2));
  PairAI p2 = PairAI.create(AInteger.create(3),AInteger.create(4));
  PairAI p3 = PairAI.create(AInteger.create(5),AInteger.create(6));
  ListPairAI sample = ListPairAI.cons(p1,
                        ListPairAI.cons(p2,
                          ListPairAI.cons(p3, ListPairAI.empty())));

  printElements(sample.funcIterator());

  PairAI zero2 = PairAI.create(AInteger.create(0),AInteger.create(0));
  PairAI sum = sumElements(zero2, sample.funcIterator());
  System.out.println("Sum = " + sum);
```

with output:

```
  Element = (1,2)
  Element = (3,4)
  Element = (5,6)
  Sum = (9,12)
```

Voilà. Code reused.