| Object-Oriented Design | Lecture 11 |
|---|---|
| CS 3500 Fall 2009 (Pucella) | Tuesday, Oct 20, 2009 |

# 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 `current()` 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. Can we figure out a nice way to reuse the interface?

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 current ();
  public FuncIterator<T> advance ();
}
```

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. When it actually comes time to use such an interface, you *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]

Suppose we wanted to have the `List` class use the above interface for its iterator. Here is the resulting implementation:

```
/* ABSTRACT CLASS FOR LISTS */
```

[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 class List {

  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<Integer> getFuncIterator ();
}


/* 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<Integer> getFuncIterator () {
    return new EmptyFuncIterator();
  }
}
```

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

  public EmptyFuncIterator () {}

  public boolean hasElement () {
    return false;
  }

  public Integer current () {
   throw new java.util.NoSuchElementException
                          ("list is empty during iteration");
  }

  public FuncIterator<Integer> advance () {
   throw new java.util.NoSuchElementException
                          ("list is empty during iteration");
  }
}


/* 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> getFuncIterator () {
    return new ConsFuncIterator(firstElement,
                                restElements.getFuncIterator());
  }
}


/* 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 current () {
      return new Integer(this.currentElement);
  }

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

In other words, nothing special, aside from the explicit conversion from `int` to `Integer` in method `current()` 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 iterator:

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

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

or a function that counts the number of elements supplied by a functional iterator:

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

    while (temp.hasElement()) {
      count++;
      temp = temp.advance();
    }
  }
```

Polymorphic interfaces are extensively used in the Java Collections framework.

## Polymorphic Methods

Adding parameterization to classes 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.

Look at functional iterators. We have functional iterators subclassing `FuncIterator<Integer>` such as above for lists of integers, and functional iterators subclassing `FuncIterator<Artifact>` such as we have on the homework. Now look at the function `countAll` I gave above, which reports the number of elements supplied by a functional iterator. Note the type of `countAll`: it takes a functional iterator that yields `Integer`s. What if we wanted to count elements supplied by a functional iterator that yields `Artifact`s, like the one in your homework? We would have to write a different `countAll` method that takes a functional iterator of type `FuncIterator<Artifact>`. Write it. What do you notice immediately?

That's right, if you write it up correctly, you'll notice that the two `countAll` methods are exactly the same, except for the type of their argument! Indeed, `countAll` does not actually care what the type returned by the functional iterator is, it doesn't do anything with it. So `countAll` looks the same no matter what type of values is provided by the functional iterator. That's wasteful. We have to write the same code over and over again, and that's error prone and difficult to maintain.

A simpler example of this sort of code duplication is with the good old identity function, which does nothing but return its argument. Here is the identity function for integers:

```
  public static Integer identity (Integer val) {
    return val;
  }
```

Here is the identity function on Booleans:

```
  public static Boolean identity (Boolean val) {
    return val;
  }
```

Here is the identity function on artifacts:

```
  public static Artifact identity (Artifact val) {
    return val;
  }
```

The type of each of those operations is `Integer` $\rightarrow$ `Integer`, `Boolean` $\rightarrow$ `Boolean`, and `Artifact` $\rightarrow$ `Artifact`. Ideally, we would like to be able to write a single function `identity`, whose type emcompasses all of these. Intuitively, you want a function `identity` that takes a value of type `T` and returns a value of type `T`, for *any* type `T`. Using a first-order logic formulation, we would like an identity function whose type is $\forall$`T.T` $\rightarrow$ `T`. Technically, a function with this type is called a parametrically-polymorphic function. (We will usually drop the "parametrically" bit, but it's kind of important for precision. There are others kinds of polymorphism out there.)

So how do we write such an identity function in Java?

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

The signature has that extra `<T>` at the front, before the return type. This is Java-speak for $\forall$`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` 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 variable that depend on that type — see example below.

Using polymorphism, we can write the single `countAll` functions that counts the number of elements given by a functional iterator no matter what type of elements that iterator yields or a function that prints the elments of a functional iterator no matter what type of elements that iterator yields:

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

  while (temp.hasElement()) {
    count++;
    temp = temp.advance();
  }
}

public static void printAll (FuncIterator<T> it) {
  FuncIterator<T> temp = it;

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

The above examples illustrate that *polymorphic methods are a way to reuse client code,* by only requiring you to write a single client method to use some code that has a polymorphic inteface.

## 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 sumAll (int initial, FuncIterator<Integer> it) {
  FuncIterator<Integer> temp = it;
  int total = initial;

  while (temp.hasElement()) {
    total = total + temp.current();
    temp = temp.advance();
  }

  return total;
}
```

(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 types 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 does have a notion of addition*. So can we get that kind of code reuse? Yes, using bounded polymorphism. Basically, bounded polymorphism lets us say "for all types *that satisfy a certain property*." That property will be 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 any subclass of that class will have that operation, meaning 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 these are left as an exercise.)

Let's modify the `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> getFuncIterator ();
}


/* 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> getFuncIterator () {
    return new EmptyFuncIterator();
  }
}


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

  public EmptyFuncIterator () {}

  public boolean hasElement () {
    return false;
  }

  public AInteger current () {
   throw new java.util.NoSuchElementException
                          ("list is empty during iteration");
  }

  public FuncIterator<AInteger> advance () {
   throw new java.util.NoSuchElementException
                          ("list is empty during iteration");
  }
}


/* 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> getFuncIterator () {
    return new ConsFuncIterator(firstElement,
                                restElements.getFuncIterator());
  }
}


/* 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 current () {
     return AInteger.create(currentElement);
  }

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

I can write a *bounded polymorphic* function `sumAll` 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 sumAll (T init, FuncIterator<T> it) {
  FuncIterator<T> temp = it;
  T total = init;

  while (temp.hasElement()) {
    total = total.add(temp.current());
    temp = temp.advance();
  }

  return total;
}
```

Note that parameterization `<T extends Addable<T>>`, read "for all types `T` that are subclasses of `Addable<T>`", that is for all types `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.

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 = sumAll(zero, sample.getFuncIterator());
System.out.println("Sum = " + sum);
```

and get

```
Sum = 6
```

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()));
  }
}
```

Instead of defining a new class for lists of pairs (we'll see a nicer way to do just that below), let me go another route and define a *functional iterator transformer*, that takes a functional iterator for integers, and wraps around it a functional iterators for pairs of integers, that takes an iterator yield $a_0, a_1, a_2, \ldots,$ and gives back an iterator yielding $(a_0, 2a_0), (a_1, 2a_1), (a_2, 2a_2), \ldots.$ This kind of iterator transformation is sometimes quite useful.

```
public class DoublingIterator implements FuncIterator<PairAI> {

  FuncIterator<AInteger> underlying;

  private DoublingIterator (FuncIterator<AInteger> u) {
    underlying = u;
  }

  public static DoublingIterator create (FuncIterator<AInteger> u) {
    return new DoublingIterator(u);
  }

  public boolean hasElement () {
    return underlying.hasElement();
  }

  public PairAI current () {
    return PairAI.create(underlying.current(),
                         underlying.current().add(underlying.current()));
  }

  public FuncIterator<PairAI> advance () {
    return create(underlying.advance());
  }
}
```

We can use this to wrap around a `FuncIterator<AInteger>` and print and sum the results:

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

  FuncIterator<PairAI> iterator =
    DoublingIterator.create(sample.getFuncIterator());

  printAll(iterator);

  AInteger zero = AInteger.create(0);
  PairAI zero2 = PairAI.create(zero,zero);
  PairAI sum = sumAll(zero2, iterator);
  System.out.println("Sum = " + sum);
```

with output:

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

Note that we get to reuse both the `printAll` and `sumAll` functions — win!

## Polymorphic Classes

There is a final bit of parameterization that I have not yet mentioned, and that becomes clear when we look at the above examples. Polymorphic interfaces let us reuse code when writing an interface. What about defining polymorphic classes? This would be useful certainly for lists, which should be defined for some type `T` of underlying values, instead of fixing a type such as `Integer` or `AInteger` in the definition. A class can be parameterized just like an interface, using a similar declaration.

Here is the definition of `List<A>`, a parameterized version of `List`, with the following signature, and the same specification as earlier:

```
CREATORS     List<A> empty ()
             List<A> cons (A, List<A>)

ACCESSORS    boolean isEmpty ()
             A first ()
             List<A> rest ()
             String toString ()
```

(I'm dropping functional iterators for now.)

Following the design pattern we have for deriving an implementation from an ADT, we get the following code:

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

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

  public static <A> List<A> cons (A i, List<A> l) {
    return new ConsList<A>(i,l);
  }

  public abstract boolean isEmpty ();

  public abstract A first ();

  public abstract List<A> rest ();

  public abstract String toString ();
}


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

  public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }

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

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

  public String toString () {
      return "";
  }
}


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

  private A firstElement;
  private List<A> restElements;

  public ConsList (A f, List<A> r) {
    firstElement = f;
    restElements = r;
  }

  public boolean isEmpty () {
    return false;
  }

  public A first () {
    return firstElement;
  }

  public List<A> rest () {
    return restElements;
  }

  public String toString () {
      return firstElement.toString() + "   " + restElements.toString();
  }
}
```

Couple of things to notice. First off, to invoke a constructors for a parameterized class, you need to supply the type at which you want to instantiate the class. For instance, `new EmptyList<Integer>()`. If you forget that, the system will essentially use `Object` as a type, which generally will *not* do what you want.

The other thing to notice is that static methods are polymorphic. That's needed because of the way Java deals with polymorphic classes. More specifically, the way Java deals with type parameters—they are technically associated with an instance of a class, and because what is associated with an instance of a class is not accessible from static elements in the class, the static methods in a class cannot refer to the parameter. Meaning that in order to write a creator `cons()` that takes an element of type `A` and a list of `A`s and returns a list of `A`s, we need to say that the type of that creator is: for all types `A`, the `cons` creators takes an `A` and a `List<A>` and produces a `List<A>`, which gives us the above method definitions.[2]

This means, in particular, that the type variable used in a static method has nothing to do with the type parameter in the class implementing that static method! We could have written the abstract class as follows, and it would still work:

[2]Here is an explanation, if you're curious. In Java, the code for a polymorphic class is not actually duplicated, there is really only one definition of `List` around, and so the type parameter of a polymorphic class is thought of as kind-of-special field, And fields in an object are not visible from static methods in the class. And indeed, you should be able to invoke `empty` even if you have no lists around. Another consequence of the Java handling of polymorphism is that type parameters, as soon as type checking is done, do not actually exist at runtime. This means, in particular, that we cannot use a type argument in places where the type would have a runtime existence, such as in a cast; uses such as `T x = (T) foo` are disallowed, as well as `instanceof` checks.)

```
public abstract class List<A> {

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

  public static <U> List<U> cons (U i, List<U> l) {
    return new ConsList<U>(i,l);
  }

  public abstract boolean isEmpty ();

  public abstract A first ();

  public abstract List<A> rest ();

  public abstract String toString ();
}
```

`Exercise:` *Add a method* ***`getFuncIterator()`*** *to the above signature, with type* ***`FuncIterator<A> getFuncIterator ()`****, where* ***`A`*** *is the type variable for class* ***`List<A>`****. Implement those functional iterators — which should be implemented similarly as for class* ***`List`*** *earlier in the lecture, except with polymorphic classes.*

As another example of a polymorphic class (that uses bounded polymorphism!), here is the definition of addable pairs, that is, pairs of elements of two types. Such pairs are addable when the types themselves are addable, since to add two pairs, we add the respective components of the pairs. We can specify such a constraint on the parameterization of a class using the same kind of bounded polymorphism we saw earlier:

```
public class APair<T extends Addable<T>,U extends Addable<U>>
        implements Addable<APair<T,U>> {

  private T first;
  private U second;

  private APair(T f, U s) {
    first = f;
    second = s;
  }

  public static <T,U> APair<T,U> create (T f, U s) {
    return new APair<T,U>(f,s);
  }

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

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

  public APair<T,U> add (APair<T,U> val) {
    return create((this.first().add(val.first()), this.second().add(val.second()));
  }

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

Now, we can create lists of pairs of addable integers, using a declaration such as:

```
  APair<AInteger,AInteger> onetwo =
    APair.create(AInteger.create(1),AInteger.create(2));
  APair<AInteger,AInteger> threefour =
    APair.create(AInteger.create(3),AInteger.create(4));

  List<APair<AInteger,AInteger>> e = List.empty();
  List<APair<AInteger,AInteger>> s = List.cons(onetwo,List.cons(threefour,e));
```

and if you define `getFuncIterator()` correctly for polymorphic lists above, you automatically get to reuse our old iterator functions:

```
  printAll(s.getFuncIterator());
  APair<AInteger,AInteger> sum = sumAll(s.getFuncIterator());
  System.out.println("Sum = " + sum);
```

with output:

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

as expected.