Generic Methods

A couple of lectures ago, we introduced generic interfaces, that is, interfaces of the form `FuncIterator<Integer>` that are *parameterized* by a type:

```
public interface FuncIterator<T> {
  public boolean hasElement ();
  public T current ();
  public FuncIterator<T> advance ();
}
```

We saw that this was 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.

To maximize code reuse in the presence of generic interfaces, however, we need a bit more than what we have seen until now.

Suppose that I want you to write, in some class `IteratorUtils`, a method that reports how many items can be provided by a given functional iterator. That is, I want a method that, when given a functional iterator, tells you how many elements it can provide. The obvious way to do that is just to repeatedly advance the iterator until it has no more elements. Let's make the method static. (Something for you to think about: why?)

```
...

public static int countElements (FuncIterator<Integer> it) {
  FuncIterator<Integer> tmp = it;
  int count = 0;
  while (hasElement(tmp)) {
    count = count + 1;
    tmp = tmp.advance();
  }
  return count;
}
```

This code can be made shorter, of course, but I aim for clarity right now. Most importantly, note that type of `countElements`: it takes a functional iterator that supplies integers. What

if we wanted to count elements supplied by a functional iterator that gives back `Line`, like the one in your homework? We would have to write a different `countElements` methods that can accept a functional iterator of type `FuncIterator<Line>`. Write it. What do you notice immediately?

That's right, if you write it up correctly, you'll notice that the two `countElements` methods are exactly the same, except for the type of their argument! Indeed, `countElements` do not actually care what the type returned by the functional iterator is, it doesn't do anything with it. So `countElements` 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, as we know.

(Exercise: can you think of an operation on functional iterators that does not have that property—for instance, an operation on functional iterators that only makes sense if the functional iterator provides integers?)

So how could we write `countElements` so that we don't have to write multiple copies for different types? Thinking about it, what you really want to say is that `countElements` takes an argument of type `FuncIterator<T>` for *any* `T`. That's what a generic method will let you express.[1] Here is the method as you could write it:

```
...

public static <T> int countElements (FuncIterator<T> it) {
  FuncIterator<T> tmp = it;
  int count = 0;
  while (hasElement(tmp)) {
    count = count + 1;
    tmp = tmp.advance();
  }
  return count;
}
```

Things to note: the signature has that extra `<T>` at the front, before the return type. That's the indication that it's a generic method, and the `T` in the angle brackets indicates what is the *type variable* that you are using in the method definition. You can read `<T>` as: "for any type `T`..." The `T` can be used in the types of the signature, and also in the body of the method. (Because we want the local variable `tmp` to have the same type as the argument, we declare it at type `FuncIterator<T>` in the first line of the method.)

In summary: *generic methods are a way to reuse client code.* (Only requiring you to write a single client method to use some code that has a generic inteface.)

[1] Generic methods are also sometimes called parametrically-polymorphic methods, but that's a mouthful.

Generic Classes

Generic interfaces, that is, interfaces that are parameterized by a type. They let us reuse code when writing an interface. (We only have to write one parameterized interface instead of multiple interfaces differing only by a type.)

What about defining generic classes? A class can be parameterized just like an interface, using a similar declaration. As a simple example, consider the following (almost useless) class:

```
public class Wrapper<T> {
  private T content;
  public Wrapper (T v) {
    content = v;
  }
  public T getWrappedContent () {
    return this.content;
  }
}
```

The `T` appearing in the class declaration is a parameter to the class declaration. Think of it as an argument to a method, except it's an argument to a class. Every other occurrence of `T` we can think of as being replaced by the argument supplied to the class when we instantiate it. These occurrences of `T` can occur at pretty much all the places where a normal type can be used. (There are some exceptions, which I will return to below.)

To use the class, you need to specify a type for the type parameter. That type is required to be either a class type, interface type, or an array. For example, this creates a new `Wrapper` instance around integers:

```
Wrapper<Integer> w = new Wrapper<Integer>(10);
```

Note that we *need to add the type argument to the constructor* as well.

(Work it out—`Wrapper<Integer>` can be thought of as the class definition where every occurrence of `T` is replaced by `Integer`. This is a good working model, but be careful with the details; the code is not actually duplicated, there is really only one definition of `Wrapper` around, and the types, 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.)

Let's look at a slightly more interesting example, stacks. The way we have defined them, stacks can only contain integers. But that's hardly general. And moreover, there is nothing really specific about integers in the stacks we defined. The same implementation can handle stacks with arbitrary content. It makes sense to parameterize the stack implementation by the type of stack content.

Here is the code produced by the recipe we saw in class (not the one with inheritance—we're not dealing with inheritance here):

```
public abstract class Stack<T> {

  public static <U> Stack<U> emptyStack () {
    return new EmptyStack<U>();
  }

  public static <U> Stack<U> push (Stack<U> s, U i) {
    return new PushStack<U>(s,i);
  }

  public abstract boolean isEmpty ();
  public abstract T top ();
  public abstract Stack<T> pop ();
}


// Concrete subclass for empty creator

class EmptyStack<T> extends Stack<T> {
  public EmptyStack () { }

  public boolean isEmpty () { return true;  }

  public T top () {
    throw new IllegalArgumentException("EmptyStack<T>,top()");
  }

  public Stack<T> pop () {
    throw new IllegalArgumentException("EmptyStack<T>.pop()");
  }
}


// Concrete subclass for push creator
```

```
class PushStack<T> extends Stack<T> {
  private T topVal;
  private Stack<T> rest;

  public PushStack (Stack<T> s, T v) {
    topVal = v;
    rest = s;
  }

  public boolean isEmpty () { return false;  }

  public T top () { return this.topVal; }

  public Stack<T> pop () { return this.rest; }
}
```

Most of it is exactly as you would expect. Instead of working with `Stack`, we work with `Stack<T>`, where `T` stands for the content of the stack, and of course the concrete subclasses also are parameterized by the type of stack content. Note that `PushStack<T>` is a subclass of `Stack<T>`—the same `T`.

The only subtlety here is the static methods in the abstract class `Stack<T>`. Because of the way Java implements generics (this is an implementation detail that propagates up to the language design level, never a good thing because it tends to introduce "special cases" to worry about), the type parameter of a generic class is thought of as kind-of-special a field, And fields in an object are not accessible from a static method of the class. (Partly, because a static method needs to make sense even if you don't have any instances of the class around!) Here too. You should be able to invoke `empty` even if you have no stack around.

Look at the static method `push`. What does it really promise? It promises that if you give it some stack `Stack<U>` for some type `U`, and a value of type `U`, it will give you back a new stack of that same type `Stack<U>`. This is exactly what generic methods are meant to capture, and this is why we use generic methods here for creators. Thus, if `s1` is a stack of Booleans (type `Stack<Boolean>`), then the method invocation:

```
Stack<Boolean> s2 = Stack.push(s1,true);
```

you can think of as executing as follows: when `Stack.push` executes, type variable `T` in the static method gets bound to `Boolean`, and if you look at the code, will end up invoking the `PushStack<Boolean>` constructor, creating a new stack of Booleans, as requested.

(Possible point of confusion: Note that the type variable occurring in the static method is different than the type variable declared at the top of the class. That's an unfortunate

source of confusion. We are parameterizing two things. At one level, we are parameterizing the class `Stack` to allow different types of stacks. At another level, we are parameterizing the static methods to allow them to work with parameterized stacks. In particular, because the type parameters involved in the class and in the static methods play different roles, they need not have the same name. Meditate on this.)

Exercise: (for you to think about) Try to come up with a "modified" recipe for working with generic ADTs (that is, ADT which are parameterized by a type, such as a stack ADT that can work with arbitrary content type). The recipe for the generic stack ADT should give you back the code above.