Generics in General

Last class, we introduced generic interfaces, that is, interfaces that are parameterized by a type.

Today, let's try to generalize this to parametrizing all classes, not just interfaces.

A class can be parameterized just like an interface, using a similar declaration. For instance, consider the following (really useless) class:

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

The `T` appearing in the class declaration is a binder—think of it as a parameter to a method. 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 reference type (class or array, including interfaces) for the type parameter. For instance:

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

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

(Work it out—`SimpleWrapper<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 `SimpleWrapper` 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.)

Couple of interesting bits are worth discussing. First off, it is possible to constrain the type parameter to be a subclass of something specific. For instance, suppose we wanted to constrain the `SimpleWrapper` class to subclasses of `Salaried`, perhaps because we want to invoke the `getSalary()` method on the wrapped value.

```
public class SimpleWrapper<T extends Salaried> {
  private T content;
  public SimpleWrapper (T v) {
    content = v;
  }
  public T getContent () {
    return this.content;
  }
  public int getSalary () {
    return this.content.getSalary();
  }
}
```

(Question for you to ponder: why is this different from `SimpleWrapper<Salaried>`?)

We can do a lot of useful, interesting things with such constraints. For instance, suppose we wanted to have `SimpleWrapper` implement the `Comparable` interface, so that we can compare a `SimpleWrapper` object to another `SimpleWrapper` object. Moreover, we want the comparison to depend on the underlying values in the wrappers. In order to do this, we need to have the parameter `T` itself implement (that is, subclass) the `Comparable<T>` interface. Once that is specified, we can have the class implement `Comparable<SimpleWrapper<T>>`, to allow comparisons with other `SimpleWrapper<T>` objects. Here is the code:

```
public class SimpleWrapper<T extends Comparable<T>>
                        implements Comparable<SimpleWrapper<T>> {
  private T content;
  public SimpleWrapper (T v) {
    content = v;
  }
  public T getContent () {
    return this.content;
  }
  public int compareTo (SimpleWrapper<T> that) {
    return (this.content.compareTo (that.getContent()));
  }
}
```

Take some time to make sure you understand the above code!

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 (modulo the type declarations) can deal with stacks of arbitrary content. (Indeed, it is a simple matter to define stacks to hold `Objects`.) It makes sense to parameterize the stack implementation by the type of stack content. There are some subtleties, however, due to the way we have implemented the ADT. Here is the code:

```
public abstract class Stack<T> {
  public static <T> Stack<T> emptyStack () {
    return new EmptyStack<T> ();
  }
  public static <T> Stack<T> push (Stack<T> s, T i) {
    return new PushStack<T> (s,i);
  }
  public abstract boolean isEmpty ();
  public abstract T top ();
  public abstract Stack<T> pop ();
}

class EmptyStack<T> extends Stack<T> {
  public EmptyStack () { }
  public boolean isEmpty () {
    return true;
  }
  public T top () {
    throw new IllegalArgumentException ("Invoking top() on empty stack");
  }
  public Stack<T> pop () {
    throw new IllegalArgumentException ("Invoking pop() on empty stack");
  }
}

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;
  }
}
```

It's all fairly straightforward, except for the static methods. Part of the problem is that static methods live in the class, while the type parameter you can think of as living with the object. (Type parameter `T` is like an instance value, and therefore not accessible from static methods.)

Because static method are not part of an object, there is no implicit type `T` at which they work. (For a dynamic method, they belong to a specific object when invoked, and that object "remembers" the type `T` that was instantiated when the object was created in the first place. Thus, a `Stack<Integer>` object remembers that it is a stack that was created at type `Integer`.)

In fact, look at the static method `push`. What does it really promises? It really promises that if you give it some stack `Stack<X>` for some type `X`, and a value of type `X`, it will give you back a new stack of that same type `Stack<X>`. It is a so-called *generic method* (also sometimes called a parametrically polymorphic method, but that's a mouthful). To indicate that it is a generic method, that is, a method that can work at multiple types, we use an indication at the beginning of the method definition that highlights the type variables used in the method declaration. Thus, we get the method declaration:

```
public static <T> push (Stack<T> s, T i) {
  ...
}
```

and we can use the type variable in the body of the static method, which is bound to the type at which the method is invoked. (In this case, depending on the arguments passed to the static method.) Thus, if `s` is a stack of Booleans (type `Stack<Boolean>`), then the method invocation:

```
Stack<Boolean> new_s = Stack.push (s,true);
```

When `Stack.push` executes, 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.

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. In fact, we could as well write the following in our `Stack<T>` class above:

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

This still says what it should: give me any stack (with stacked values of type `U`, and a value of type `U`, and I will give you a new stack of `U`s.