Introduction to Generics in Java 5

One trouble with the list, stack, and queue ADTs that we have written so far is that the type of the objects stored in them cannot be checked at compile time. For example, consider the code:

```
// Store a queue of Strings for people's names.
Queue nameQ = new Queue(10);
...
nameQ.enqueue("allan");
nameQ.enqueue("brenda");
nameQ.enqueue(new Integer(1));
nameQ.enqueue(2);  // 2 "autoboxed" as new Integer(2)
...
...
// Look up string names in student database db
while (nameQ.size() > 0) {
  Student s = db.find((String) nameQ.dequeue());
  /* previous line generates a ClassCastException
  at runtime when the Integer objects are dequeued */
}
```

We only find out at runtime that somewhere a non-String was stored in the queue.

Generics allow this type of error to be caught at compile-time.

Beware: The DrJava interaction pane does not treat generics the same as the compiler does.

Parameterized Types

Java 5 has introduced generics (also called parameterized types) to catch such errors at compile-time. Two key questions:

1. How do you use generics?
2. How do you define classes and interfaces to use generics?

1. Usage: The use of `<String>` specifies a parameterized type, as in:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Test {

  public static void main(String[] args) {
   // Store a list of integers.
   ArrayList<String> nameQ = new ArrayList<String>(10);
   nameQ.add("allan");  // Checked type ok.
   nameQ.add("brenda"); // Checked type ok.Whoops again
   nameQ.add(new Integer(99432));  // Whoops.
   nameQ.add(9234);   // Whoops again.
   // ...
   Iterator<String> it = nameQ.iterator();
   while (it.hasNext()) {
     String s = it.next(); // Don't need to cast.
   }
  }
}
```

Both lines containing "Whoops" generate compile-time errors. They are therefore easy to catch and fix. Also, the code is cleaner without the need to cast items back to the expected type.

Declaring ADTs with Parameterized Types

1. Declaration: To allow `Queue` to make use of generics, we can change that interface to:

```
public interface Queue<E> {
  void enqueue(E o);
  E dequeue();
  E head();
  int size();
  int capacity();
}
```

Here E denotes any non-primitive type (aka reference type). The convention is to use a single capital letter such as E or T.

Defining an ADT with Parameterized Types

1. Declaration (cont): Now `ArrayQueue.java` could be defined as follows (we have left off all but the essentials to help focus on the new parts):

```
public class ArrayQueue<T>
                    implements Queue<T> {
  T[] contents;
  int size;

  public ArrayQueue(int n) {
    contents = (T[]) new Object[n];
    // Previous line is ok, but generates a
    // compile time warning about an unchecked cast.
  }

  public void enqueue(T o) { contents[size++] = o; }

  public T dequeue() {
    T v = contents[0];
    size--;
    for (int i = 0; i<size; i--)
      contents[i] = contents[i+1];
    return v;
  }
  public T head() { return contents[0]; }
  public int size() { return size; }
  public int capacity() { return contents.length;}
}
```

Here T denotes the parameterized type. Note Java 5 does not allow the creation of arrays of parameterized types, instead a cast is needed in ArrayQueue's constructor. Unfortunately, this cast raises a compile-time warning (but not an error).

Using a Queue with Parameterized Types

2. Usage (one last time): Here we use our new `ArrayQueue` class:

```
public class TestGenericQueue {
  public static void main(String[] args) {
   // Store a queue of Strings.
   Queue<String> nameQ = new ArrayQueue<String>(10);

   nameQ.enqueue("allan");  // Checked type ok.
   nameQ.enqueue("brenda"); // Checked type ok.
   // nameQ.enqueue(new Integer(99432));  // Whoops.
   // nameQ.enqueue(9234);   // Whoops again.
   // ...

   while (nameQ.size()>0) {
     String s = nameQ.dequeue(); // Look, Ma, no cast.
     System.out.println(s);
   }
  }
}
```

The type `String` can be replaced by any reference (i.e. non-primitive) type such as `Integer`, or a generic type, say `ArrayList<String>`.

The lines containing “Whoops” would cause a compile-time error if they were uncommented, as desired.

The only downside here is the generation of the “unchecked cast” warning when `ArrayQueue.java` is compiled on the line the array is created. It is a warning, not an error, and it is unavoidable in Java 5.