CSE 341 - Programming Languages - Autumn 2012 Java Generics & Nested Classes

Nested Classes in Java

Class and interface definitions can be nested inside of other classes in Java. Some interesting issues arise:

- What are the visibility rules for nested classes or interfaces?
- Can code in the nested class reference variables in enclosing scopes? For example, can an instance of a nested class reference the non-static fields (instance variables) of its enclosing class?
- What is the lifetime of an instance of a nested class? If it was created in the context of an instance of the enclosing class, can it outlive the enclosing instance?

Java's Approach

Classes and interfaces can be nested inside of other classes. The simplest case is when the class is declared as *static*. In that case, it can be accessed in the same way as a static field; you don't need to have an instance of the enclosing class.

Example: `java.awt.geom.Rectangle2D.Double`

Here `Double` is a static nested class of `Rectangle2D`. Primary benefit: name space control. Interfaces must always be static.

More interesting case: non-static nested classes (called an *inner class*). An instance of the inner class can access non-static fields of the enclosing class, as well as static ones. Such an instance can only be created in the context of a particular instance of the enclosing class -- you can't get at the class otherwise. One instance of the outer class can have no associated instances of the inner class, or several.

The iterator class in MyArray.java is a good example of an inner class. Its instances have access to the state of the MyArray instance. Furthermore, we can have several iterators associated with a given MyArray instance. As long as an iterator is accessible (not available for garbage collection), then its MyArray instance can't be garbage collected either, even if there are no other references to the MyArray instance.

Anonymous Inner Classes

Java allows one to declare anonymous inner classes. These can be useful for simple classes, such as iterators or event listeners. Although the language allows you to have arbitrarily complex anonymous classes, good style is to keep them very simple -- if it starts getting complicated make a named inner class. (Personally I tend to avoid them altogether, but this is a matter of taste.)

Example: MyArray3.java. Here is the code:

```
// variant of MyArray, using an anonymous inner class for the iterator


import java.util.Iterator;
import java.util.NoSuchElementException;
import java.awt.Point;

public class MyArray3<E> {
    /* internal array to actually hold the data */
    private E [] internalArray;

    // the constructor
    public MyArray3 (int n) {
        internalArray = (E[]) new Object[n];
        // unfortunately this doesn't work due to current Java limitations:
        // internalArray = new E[n];
    }

    public int length() {
        return internalArray.length;
    }

    // get an element
    public E at(int i) {
        return internalArray[i];
    }

    // set an element
    public void set(int i, E value) {
        internalArray[i] = value;
    }

    // return an iterator for this array
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int index = 0;
            public E next() {
                E temp;
                if (!hasNext())
                    throw new NoSuchElementException("no next element available");
                temp = internalArray[index];
                index++;
                return temp;
            }
            public boolean hasNext() {
                return index < internalArray.length;
            }
            public void remove() {
                /* this is an optional operation - we don't support it */
                throw new UnsupportedOperationException("remove operation not supported");
            }
        };
    }

    public static void main(String[] args) {
        MyArray3<Integer> a = new MyArray3<Integer>(2);
        a.set(0, new Integer(50));
        a.set(1, new Integer(100));
        // we can have several iterators active at once on an instance
        // of MyArray
        Iterator<Integer> iter1 = a.iterator();
        Iterator<Integer> iter2 = a.iterator();
        while(iter1.hasNext()) {
            System.out.println(iter1.next());
        }
        while(iter2.hasNext()) {
            System.out.println(iter2.next());
        }
    }
}
```

References

- tutorial on Java Nested Classes