

CITS2200
Data Structures and Algorithms

Cara MacNish

School of Computer Science & Software Engineering
University of Western Australia

Topic 8

Objects and Iterators

- Generalising ADTs using objects
 - wrappers, casting
- Iterators for Collection Classes
- Inner Classes

Reading: Lambert & Osborne, Sections 6.3–6.5;
2.3.5

8.1 Generalising ADTs to use Objects

Our ADTs so far have stored primitive types.

eg. block implementation of a queue from Section 5

```
public class QueueCharBlock {  
  
    private char[] items;  
    private int first, last;  
  
    public char dequeue() throws Underflow {  
        if (!isEmpty()) {  
            char a = items[first];  
            first++;  
            return a;  
        }  
        ...  
    }  
}
```

This queue will **only** work for characters. We would need to write another for integers, another for a queue of strings, another for a queue of queues, and so on.

Far better would be to write a single queue that worked for **any** type of object.

In object-oriented languages such as Java this is easy, providing we recall a few object-oriented programming concepts from Section **2.4**

— inheritance, casting, and wrappers.

8.1.1 Objects in the ADTs

The easiest part is changing the ADT. (The more subtle part is using it.)

Recall that:

- **every** class is a subclass of the class `Object`
- a variable of a particular class can hold an **instance** of **any subclass** of that class

This means that if we define our ADTs to hold things of type `Object` they can be used with objects from **any other class**!

```
/**
 * Block representation of a queue (of objects).
 */
public class QueueBlock {

    private Object[] items;           // array of Objects
    private int first;
    private int last;

    public Object dequeue() throws Underflow { // returns an Object
        if (!isEmpty()) {
            Object a = items[first];
            first++;
            return a;
        }
        else throw new Underflow("dequeuing from empty queue");
    }
}
```

8.1.2 Wrappers

The above queue is able to hold any type of object — that is, an instance of any subclass of the class `Object`. (More accurately, it can hold any reference type.)

But there are some commonly used things that are not objects — the primitive types.

In order to use the queue with primitive types, they must be “wrapped” in an object.

Recall from Section 2.4 that Java provides wrapper classes for all primitive types.

Autoboxing — Note for Java 1.5

Java 1.5 provides **autoboxing** and **auto-unboxing**. Effectively acts as automatic wrapping and unwrapping.

```
Integer i = 5;  
int j = i;
```

However:

- Not a change to the underlying language — the **compiler** recognises the mismatch and substitutes code for you:

```
Integer i = Integer.valueOf(5)  
int j = i.intValue();
```

- Can lead to unintuitive behaviour. Eg:

```
Long w1 = 1000L;  
Long w2 = 1000L;  
if (w1 == w2) {  
    // do something  
}
```

may not work. Why?

- Can be slow. Eg. if a, b, c, d are Integers, then

```
d = a * b + c
```

becomes

```
d.valueOf(a.intValue() * b.intValue() + c.intValue())
```

For more discussion see:

<http://chaoticjava.com/posts/autoboxing-tips/>

8.1.3 Casting

Recall that in Java we can assign “up” the hierarchy — a variable of some class (which we call its reference) can be assigned an object whose reference is a subclass.

However the converse is not true — a subclass variable cannot be assigned an object whose reference is a superclass, even if that object is a subclass object.

In order to assign back down the hierarchy, we must use **casting**.

This issue occurs more subtly when using ADTs. Recall our implementation of a queue...

```
public class QueueBlock {
    private Object[] items;           // array of Objects
    ...
    public Object dequeue() throws Underflow { // returns an Object
        if (!isEmpty()) {
            Object a = items[first];
            first++;
            return a;
        }
        else...
```

Consider the calling program:

```
QueueBlock q = new QueueBlock();
String s = "OK, I'm going in!";
q.enqueue(s);           // put it in the queue
s = q.dequeue();       // get it back off ???
```

The last statement fails. Why?

The queue holds `Object`s. Since `String` is a subclass of `Object`, the queue can hold a `String`, but its reference in the queue is `Object`. (Specifically, it is an element of an array of `Object`s.)

`dequeue()` then returns the “`String`” with reference `Object`.

The last statement therefore asks for something with reference `Object` (the superclass) to be assigned to a variable with reference `String` (the subclass), which is illegal.

We have to cast the `Object` back “down” the hierarchy:

```
s = (String) q.dequeue();           // correct way to dequeue
```

Generics — Note for Java 1.5

Java 1.5 provides an alternative approach. **Generics** allow you to specify the type of a collection class:

```
Stack<String> ss = new Stack<String>();  
String s = "OK, I'm going in!";  
ss.push(s);  
s = ss.pop()
```

Like autoboxing, generics are handled by compiler rewrites — the compiler checks that the type is correct, and substitutes code to do the cast for you.

Generics in Java are complex and are the subject of considerable debate.

Some interesting articles:

<http://www-128.ibm.com/developerworks/java/library/j-jtp01255.html>

http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consider_1.html

8.2 Iterators

It is often necessary to **traverse** a collection — look at each item in turn.

Example:

In **Lab Exercise 4** you were asked to get characters out of a basic `LinkedListChar` one at a time and print them on separate lines. Doing this using the supplied methods destroyed the list.

We now know this to be the behaviour of a `Stack`, which has no public methods for accessing items other than the top one.

Example:

In Chapter 3 we developed the simple linked list class. In order to print out the items in the list (without destroying it) we provided the following `toString` method:

```
public String toString () {
    LinkChar cursor = first;
    String s = "";
    while (cursor != null) {
        s = s + cursor.item;
        cursor = cursor.successor;
    }
    return s;
}
```

This is not a generic approach. If we wanted to look at the items for another purpose — say to print on separate lines, or search for a particular item — we would have to write another method using another loop to do that.

A more standard, generic approach is to use an **iterator**.

An iterator is a companion class to a collection (known as the iterator's **backing collection**), for traversing the collection (ie examining the items one at a time).

An iterator uses standard methods for traversing the items, independently of the backing collection. In Java these methods are specified by the **Iterator** interface in `java.util`.

These are:

- `boolean hasNext()` — return `true` if the iterator has more items
- `Object next()` — if there is a next item, return that item and advance to the next position, otherwise throw an exception
- `void remove()` — remove from the underlying collection the last item returned by the iterator. Throws an exception if the immediately preceding operation was not `next`.

Note: some iterators do not provide this method, and throw an `UnsupportedOperationException` (arguably a poor use of interfaces).

The underlying collection must also have a method for “spawning” a new iterator over that collection. In Java’s **Collection** interface this method is called `iterator`.

8.2.1 Using an Iterator

```
public static void main(String[] args) {
    Queue q = new QueueCyclic();
    q.enqueue(Character('p'));
    q.enqueue(Character('a'));
    q.enqueue(Character('v'));
    q.enqueue(Character('o'));
    Iterator it = q.iterator();
    while(it.hasNext())
        System.out.println(it.next());
}
```

8.2.2 Implementation — backing queue

```
import java.util.Iterator;
public class QueueCyclic implements Queue {

    Object[] items;           // package access for
    int first, last;         // companion class

    public QueueCyclic (int size) {
        items = new Object[size+1];
        first = 0;
        last = size;
    }

    public Iterator iterator() {
        return new BasicQueueIterator(this);
    }

    ...
}
```

8.2.3 Implementation — iterator

```
import java.util.Iterator;

class BasicQueueIterator implements Iterator {
    private Queue backingQ;
    private int current;

    BasicQueueIterator(Queue q) {
        backingQ = q;
        current = backingQ.first;
    }

    public boolean hasNext () {
        return !backingQ.isEmpty() &&
            ((backingQ.last >= backingQ.first && current <= backingQ.last) ||
            (backingQ.last < backingQ.first &&
            (current >= backingQ.first || current <= backingQ.last)))
    }
}
```

```
public Object next () {
    if (!hasNext())
        throw new NoSuchElementException("No more elements.");
    else {
        Object temp = backingQ.items[current];
        current = (current+1)%backingQ.items.length;
        return temp;
    }
}

public void remove () {
    throw new UnsupportedOperationException
        ("Cannot remove from within queue.");
}
}
```

8.2.4 Fail-fast Iterators

Problem: What happens if backing collection changes during use of an iterator?

eg. multiple iterators that implement `remove`

⇒ can lead to erroneous return data, or exceptions (eg null pointer exception)

One Solution: Disallow further use of iterator (throw exception) when an unexpected change to backing collection has occurred — `fail-fast` method

Changes to backing collection...

```
public class QueueCyclic implements Queue {

    Object[] items;
    int first, last;
    int modCount;           // number of times modified

    public void enqueue (Object a) {
        if (!isFull()) {
            last = (last + 1) % items.length;
            items[last] = a;
            modCount++;
        }
        else throw new Overflow("enqueueing to full queue");
    }
    ...
}
```

Changes to iterator...

```
class BasicQueueIterator implements Iterator {
    private Queue backingQ;
    private int current;
    private int expectedModCount;

    public Object next () {
        if (backingQ.modCount != expectedModCount)
            throw new ConcurrentModificationException();
        if (!hasNext())
            throw new NoSuchElementException("No more elements.");
        else {
            Object temp = backingQ.items[current];
            currentIndex = (current+1)%backingQ.items.length;
            return temp;
        }
    }
}
```

8.3 Inner Classes

From a software engineering point-of-view the way we have implemented our iterator is not ideal:

- private variables of `QueueCyclic` were given “package” access so they could be accessed from `BasicQueueIterator` — now they can be accessed from elsewhere too
- `BasicQueueIterator` is only designed to operate correctly with `QueueCyclic` (implementation-specific) but there is nothing preventing applications trying to use it with other implementations

Later versions of Java provide a tidier way... **inner classes**.

Inner classes are declared within a class:

```
public class MyClass {  
  
    // fields  
  
    // methods  
  
    private class MyInnerClass {  
  
        // fields  
  
        // methods  
    }  
  
    ...  
}
```

Cyclic queue implementation using an inner class...

```
import java.util.Iterator;
public class QueueCyclic implements Queue {

    private Object[] items;           // private again
    private int first, last;         //

    ...

    public Iterator iterator() {
        return new BasicQueueIterator(); // no "this"
    }

    private class BasicQueueIterator implements Iterator {

        private int current;

        // no need to store backing queue
    }
}
```

```

private BasicQueueIterator() { // only constructed in outer class
    current = first; // variable accessed directly
} // no passing of backing queue

public boolean hasNext () {
    return !isEmpty() && // methods & variables
        ((last >= first && current <= last) || // accessed directly
        (last < first && (current >= first || current <= last)))
}
} // end of inner class

} // end of QueueCyclic

```

Q: What other structures have we seen where the use of inner classes would be appropriate?