Objects and Iterators

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

Reading: Lambert and Osborne, Sections 6.3–6.5 and 2.3.5
1. **Generalising ADTs to use Objects**

Our ADTs so far have stored primitive types.

eg. block implementation of a queue from Topic ??.

```java
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 Topic ??.

— inheritance, casting, and wrappers.
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("dequeueing from empty queue");
    }
}
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 Topic ?? that Java provides wrapper classes for all primitive types.
Autoboxing — Note for Java 1.5

Java 1.5 provides autoboxing and auto-unboxing — effectively, automatic wrapping and unwrapping done by the compiler.

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

However:

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

```java
Integer i = Integer.valueOf(5)
int j = i.intValue();
```
• Can lead to unintuitive behaviour. Eg:

```java
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

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

becomes

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

For more discussion see:

http://chaoticjava.com/posts/autoboxing-tips/
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 \texttt{Objects}. Since \texttt{String} is a subclass of \texttt{Object}, the queue can hold a \texttt{String}, but its reference in the queue is \texttt{Object}. (Specifically, it is an element of an array of \texttt{Objects}.)

\texttt{dequeue()} then returns the “\texttt{String}” with reference \texttt{Object}.

The last statement therefore asks for something with reference \texttt{Object} (the super-class) to be assigned to a variable with reference \texttt{String} (the subclass), which is illegal.

We have to cast the \texttt{Object} back “down” the hierarchy:

\begin{verbatim}
s = (String) q.dequeue(); // correct way to dequeue
\end{verbatim}
1.4 Generics

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

```java
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.
Writing Generic Classes

/**
 * A simple generic block stack for
 * holding object of type E
 **/
class Stack<E> {

    private Object[] block;
    private int size;

    public Stack(int size) {block = new Object[size];}

    public E pop() {return (E) block[--size];}

    public void push(E el) {block[size++] = el;}
}
public static void main(String[] args){
    //create a Stack of Strings
    Stack<String> s = new Stack<String>(10);
    s.push("abc");
    System.out.println(s.pop());

    //create a stack of Integers
    Stack<Integer> t = new Stack<Integer>(1);
    t.push(7);
    System.out.println(t.pop());
}
How Generics Work

The program:

```java
Stack<String> ss = new Stack<String>(10);
String s = "OK, I’m going in!";
ss.push(s);
s = ss.pop();
```

is converted to:

```java
Stack<Object> ss = new Stack<Object>(10);
String s = "OK, I’m going in!";
ss.push(s);
s = (String) ss.pop();
```

at compile time. Generics allow the compiler to ensure that the casting is correct, rather than the runtime environment.
Some Tricks with Generics...

Note that `Stack<String>` is not a subclass of `Stack<Object>` (because you can’t put an `Integer` on a stack of `Strings`).

Therefore, polymorphism won’t allow you to define methods for all stacks of sub-classes of `String`. e.g.

```java
public int printAll(Stack<Object>);
```

Java 5 allows wildcards to overcome this problem:

```java
public int printAll(Stack<?>);
```

or even

```java
public int printAll(Stack<? extends Object>);
```
Generics in Java are complex and are the subject of considerable debate. While you may not need to write them often, it is important you understand them as the Java Collection classes all use generics.

Some interesting articles:


http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consideration_1.html

2. **Iterators**

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

**Example:**

In the lab exercises, 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 Topic ??, 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:

```java
    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`. 

© Cara MacNish & Tim French

CITS2200  Objects and Iterators  Slide 20
**Interface Iterator**<E>

The interface has the methods:

- `boolean hasNext()` — return `true` if the iterator has more items
- `E 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).
Interface Collection\textlangle}E\textrangle

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 \texttt{iterator}.

\texttt{Iterator\textlangle}E\textrangle \ \texttt{iterator()}

Have a look at the collection classes:

\texttt{http://www.csse.uwa.edu.au/programming/java/jdk1.5/api/}

Here, you will find specifications for some of the data structures we have already seen and many we are yet to discuss. You may wonder why we are bothering to implement these data structures at all!
3. Using Java Collections

Built-in (API) classes can be accessed in two ways:

1. by providing their “full name”
   ```java
   java.util.LinkedList<String> b = new java.util.LinkedList();
   ``
   Here LinkedList is a class in the API package java.util.

2. by *importing* the class
   ```java
   import java.util.LinkedList;
   ...
   LinkedList<String> b = new LinkedList();
   ```

3. You can also import all classes in a package
   ```java
   import java.util.*;
   ```
4. **java.util**

Most general data structures in the Java API are in the `util` package. There are:

1. **Collections**:
   - `LinkedList<E>`, `ArrayList<E>`, `PriorityQueue<E>`, `Set<E>`, `Stack<E>`, `TreeSet<E>`

2. **Maps**:
   - `SortedMap<K, V>`, `TreeMap<K, V>`, `HashMap<K, V>`

3. and others:
   - `Iterator<E>`, `BitSet`

These allow you to create most of the data structures you will ever need. However, it is important to be able to compare the performance and understand the limitations of each. We will also examine some data structures that are not in the Java API.
4.1 Using Iterators

The following code creates an iterator to access the elements of a queue.

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

```java
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);
    }
    ...
```
4.3 Implementation — Iterator

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

    BasicQueueIterator(QueueCyclic 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 newUnsupportedOperationException("Cannot remove from within queue.");
}
4.4 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 the 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("enqueuing to full queue");
        }
        ...
    }
Changes to the iterator...

class BasicQueueIterator implements Iterator {
    private QueueCyclic 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];
            current = (current+1)%backingQ.items.length;
            return temp;
        }
    }
}
5. Inner Classes — A Better Way to Iterate

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

Java provides a tidier way… *inner classes.*
Inner classes are declared within a class:

```java
public class MyClass {

    // fields

    // methods

    private class MyInnerClass extends MyInterface {

        // fields

        // methods
    }

    ... public MyInterface getMyInnerClass() { ... }

}```
The Inner class is able to access all of the private fields and methods of the outer class.

This gives us a very powerful method to control access to a data structure. The code of the inner class has free range over the instance variables of the outer class, but users can only access the inner class through the prescribed interface.

Inner classes are used extensively in Object Oriented Programming for call backs, remote method invocation, or listener classes.
Cyclic queue implementation using an inner class...

```java
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() { // constructed by outer class
    current = first; // variable accessed directly
} // no passing of backing queue

public boolean hasNext() {
    return !isEmpty() &&
    ((last >= first && current <= last) ||
    (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?
5.1 Foreach

Another Java 5 feature is the new control structure `foreach`. This can be used for iteration through a collection of elements. For example, the following code:

```java
int[] array = {0,2,4};
for (int i : array)
    System.out.println(i);
```

means the same as:

```java
int[] array = {0,2,4};
for (int i = 0; i < array.length; i++)
    System.out.println(array[i]);
```

But its use is not just restricted to arrays. Indeed, any object with a `iterator()` method can be used!
6. Summary

- ADTs can be made more general through the use of objects and casting
- generics provide a “cleaner” mechanism of achieving the same functionality
- iterators are a means of traversing a collection of elements
- the use of inner classes simplifies the construction of iterators