Objects and Iterators

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

Reading: Weiss Chapter 15.

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 `Objects`. 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 `Objects`.)

`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

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.

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;}
}
```

Using Generic Classes

```
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:

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

is converted to:

```
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 `String`s).

Therefore, polymorphism won't allow you to define methods for all stacks of subclasses of String. e.g.

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

Java 5 allows *wildcards* to overcome this problem:

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

or even

```
public int printAll(Stack<? extends Object>);
```

Iterators

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

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.

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`.

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<E>

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`.

```
Iterator<E> interator()
```

Have a look at the collection classes:

`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!

Using Java Collections

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

1. by providing their "full name"

   ```
   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

   ```
   import java.util.LinkedList;
   .
   .
   .
   LinkedList<String> b = new LinkedList();
   ```

3. You can also import all classes in a package

   ```
   import java.util.*;
   ```

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.

Using Iterators

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

```
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());
}
```

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);
  }
  ...
```

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 new UnsupportedOperationException
      ("Cannot remove from within queue.");
  }
}
```

Fail-fast Iterators

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

eg. multiple iterators that implement `remove`
$\rightarrow$ 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;
    }
  }
```

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:

```
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…

```
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
```

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:

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

means the same as:

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