Data Structures and Algorithms

Lecture 7b: Iterators, Generics

Department of Computing

Curtin University

Last updated: 29 March 2016

Copyright Warning

Iterators

- A LinkedList with a single ‘current’ cursor is limiting
  - Assumes only one ‘user/client’ of the LinkedList at a time
  - It would be better if every client had its own current cursor
- Iterators are designed to solve this problem
  - Each Java container (list) class has an iterator( ) method
    - » Note that it’s not *get*Iterator( ) - a bit inconsistent of Java!
  - Returns an Iterator (or more precisely, an object that inherits from the Java Iterator interface)
    - » You don’t need to know the exact object type - knowing it is an Iterator is enough (the power of interfaces and polymorphism!)
    - » Enumeration is an older Java interface for the same idea

Iterator Interface

- Quite simple; only three methods
  - hasNext( ) - queries if more items exist in the list
  - next( ) - move the cursor to the next item in the list
  - remove( ) - optional ability to remove the current item
    - » throw UnsupportedOperationException if you don't support it
  - These just give standard names to a common task
- No prev( ) - inherit from interface ListIterator for that
- Only limit is that behaviour is undefined if element removed by one client while another client iterating
  - Depends on the underlying list being iterated over
  - Some can handle this scenario (*e.g.*, linked list) others can't

Using an Iterator

– Using the iterator directly:

```
public void iterateOverList(LinkedList theList) {
   MyClass c;
   Iterator iter = theList.iterator();
   while (iter.hasNext()) {
      c = (MyClass)iter.next();          ← Get next item and cast from Object to MyClass
      doSomething(c);
   }
}
```

– Using the 'for-each' looping structure:

- Added to Java in 1.5 to simplify coding iterations
- Requires list class to implement Iterable interface

```
public void iterateOverList(LinkedList theList) {
   MyClass c;
   for (Object o : theList) {
      c =  (MyClass)o;                   ← Cast to MyClass
      doSomething(c);
}   }
```

- Also handles nested loops without causing iteration errors

Writing Your Own Iterator

– Iterators are well-suited to being implemented by private inner classes - clients only need to know about Iterator

```
public class MyLinkedList implements Iterable {      ← so for-each loop can be used. Only defines iterator() method
   ...
   public Iterator iterator() {                      ← Return a new Iterator of internal type MyLinkedListIterator
      return new MyLinkedListIterator(this);         ← Hook the iterator to this MyLinkedList object
   }

   private class MyLinkedListIterator implements Iterator {   ← Private class inside MyLinkedList
      private MyListNode iterNext;            ← Cursor (assuming MyListNode is the node class of MyLinkedList)
      public MyLinkedListIterator(MyLinkedList theList) {
         iterNext = theList.head;             ← NOTE: Able to access private field of MyLinkedList
      )
      // Iterator interface implementation
      public boolean hasNext() { return (iterNext != null) }
      public Object next() {
         Object value;
         if (iterNext == null)
            value = null;
         else {
            value = iterNext.getValue();             ← Get the value in the node
            iterNext = iterNext.getNext();           ← Ready for subsequent calls to next()
         }
         return value;
      }
      public void remove() { throw new UnsupportedOperationException("Not supported"); }
   }
}
```

Private Inner Classes

– Note that the iterator class was able to access the 'head' field of the list class
  - Even though 'head' is a private field!
  - Works because the inner class is also part of the list class
    - » The inner class has access to all private fields of the outer class
  - This is a very useful property of inner classes that are exploited in many contexts
    - » *e.g.*, event handling in GUIs, helper classes, and Iterators
    - » Replaces the terrible concept of friend classes in C++
  - You could always pass the head instead of the list though

Generics

- You'll notice in the Java 1.5/1.6 Docs that Iterator, and indeed most of the container classes, have this funny '<E>' thing appended to their class names
  - *e.g.*, Stack<E>, Iterator<E> ArrayList<E>
  - These represent what are called generic classes
    - » The same concept is in C#, and called templates in C++
- Generics allow you to parameterise the *class itself* by providing the types that the class operates on
  - *e.g.*, Stack<Plate> means "create a Stack of Plates"

Generics - Why Bother?

– When writing a container without generics, to make it general-purpose you must have Objects for fields
  - Otherwise you will be limiting what can be stored
  - But this means that the client of your container must downcast from Object to the class they actually put in
    - » Annoying, and risks a ClassCastException at runtime
  - Also, primitives are not Objects and so cannot be stored
    - » Must use the wrapper classes - Integer, Float, Double, etc
    - » Which requires *both* a downcast *and* a call to (say) .intValue( )

Generics vs Containing an Object

– Example:

```
   ArrayList squareNums = new ArrayList();
   for (int ii = 0; ii < 5; ii++)       // Set up an array of square values
      squareNums.add(new Integer(ii * ii));                  ← Must wrap int into Integer object

   int iSumOfSquares = 0;                // Go thru array and sum up the squares
   for (int ii = 0; ii < 5; ii++) {
//       iSumOfSquares += squareNums.get(ii);               ← Compiler error (int assigned an Object)
//       iSumOfSquares += ((Float)squareNums.get(ii)).intValue();     ← Run-time ClassCastException
      iSumOfSquares += ((Integer)squareNums.get(ii)).intValue();      ← Correct
   }
```

– Generics allow the client to define what type to store
- Eliminates the need for casting
  - » Code becomes much cleaner and simpler
- The compiler can ensure that everything is the right type
  - » Catches many logic errors at compile time
- Unfortunately, Java generics don’t support primitives
  - » Most other languages with generics do support primitives

Using a Generic Class

– *Using* a generic class is pretty easy - just add <type>

```
ArrayList<Integer> squareNums = new ArrayList<Integer>();
for (int ii = 0; ii < 5; ii++)        // Set up an array of square values
   squareNums.add(new Integer(ii * ii));          ← Still must wrap int into Integer object

int iSumOfSquares = 0;                // Sum up the squares
for (int ii = 0; ii < 5; ii++) {
   iSumOfSquares += squareNums.get(ii).intValue();     ← No need for cast to Integer
}
```

– There are more complicated things you can do with generics, but the basic concept remains the same

- For example: can have more than one data type parameter

  » *e.g.*, Hashtable<String, Double>: String = key/lookup type, Double = value/stored type

  code: Hashtable<String, Double> h = new Hashtable<String, Double>();

Creating a Generic Class

– *Building* a generic class is slightly different
  - It requires you to put in a ‘placeholder’ name for the data type instead of the data type itself (like a parameter)
  - The clients of your class then later provide the *real* type
  - Let’s take a hypothetical MyListNode class:

```
public class MyListNode<E> {        ← Class definition with parameter E for generic placeholder type
   public E data;                   ← Can use generic type E like a real type, just don't know exact type yet
   public MyListNode next;
}
```

  - The E (any name will do) is a placeholder for the real type that is only defined when MyListNode is actually *used*

```
public static void main() {
   MyListNode<Double> aNode;        ← E now is 'redefined' to be Double, but only for aNode
   // etc
}
```

Writing Your Own Iterator Revisited

– To make it use generics, just add **<E>**s and replace Object

```
public class MyLinkedList<E> implements Iterable<E> {
   ...
   public Iterator<E> iterator() {
      return new MyLinkedListIterator<E>(this);
   }

   private class MyLinkedListIterator<E> implements Iterator<E> {
      private MyLinkedList<E>.MyListNode<E> iterNext;
      public MyLinkedListIterator(MyLinkedList<E> theList) {
         iterNext = theList.head;
      }
      // Iterator interface implementation
      public boolean hasNext() { return (iterNext != null) }
      public E next() {
         E value;
         if (iterNext == null)
            value = null;
         else {
            value = iterNext.getValue();
            iterNext = iterNext.getNext();
         }
         return value;
      }
      public void remove() { throw new UnsupportedOperationException("Not supported"); }
   }
}
```

← Weird issue: Java generics require *full* scoping

← NOTE: No <E> in c'tor name

Writing Your Own Iterator Revisited

– …as compared to how it was back in Slide 6

```
public class MyLinkedList implements Iterable {
   ...
   public Iterator iterator() {
      return new MyLinkedListIterator(this);
   }

   private class MyLinkedListIterator implements Iterator {
      private MyListNode iterNext;
      public MyLinkedListIterator(MyLinkedList theList) {
         iterNext = theList.head;
      )
      // Iterator interface implementation
      public boolean hasNext() { return (iterNext != null) }
      public Object next() {
         Object value;
         if (iterNext == null)
            value = null;
         else {
            value = iterNext.getValue();
            iterNext = iterNext.getNext();
         }
         return value;
      }
      public void remove() { throw new UnsupportedOperationException("Not supported"); }
   }
}
```

Java Generics Under the Covers

- So how do generics actually work underneath it all?
- Java uses *type erasure*
  - The generic type E only exists for duration of compiling
    - » Temporarily replaced with real type (*e.g.*, Double) which is then used to check for type safety
  - After compilation, all the E's are converted to Object
    - » Type safety is lost at runtime – it's a compile-time-only check
    - » Explains why primitives aren't supported!

Java Generics Under the Covers

- Why did Java's creators use type erasure?
  - Backwards compatibility: they wanted to avoid having to change the .class binary format
    - » Compiling generics to Object means only Objects go in the .class
  - Unfortunately, it makes for some weird behaviour
    - » *e.g.*, Can't create arrays of generics: E[] anArray; is illegal
    - » *e.g.*, Don't even have to give the generic parameter: MyListNode and MyListNode<Double> are *both legal*
- Other languages do it differently
  - C#/C++: every used type gets its *own copy of the class*
    - » C++ actually *recompiles* the generic class *source code* each time
    - » C# is smarter: substitutes generic type within compiled code

Auto-boxing/unboxing

- Embedding a primitive into its class equivalent is referred to as ‘boxing’ in Java
  - *e.g.*, new Integer(-5) boxes -5 into an Integer object
- Java will automatically do boxing for you if possible
  - … and also does automatic unboxing too

```
Integer intObj;
int intVal;
intObj = 10;          ← Auto-boxing hides need to do intObj = new Integer(10)
intVal = intObj;      ← Auto-unboxing hides need to call intObj.intValue( )
```

  - *Slightly* offsets Java’s lack of generic support for primitives
  - Problems: 1) Slow. 2) Integer == differs from int ==
    - » Integer == is a reference comparison, int == is Integer.equals( )
    - » So try not to rely too much on auto-boxing/auto-unboxing!

Next Week

– Binary Trees