GENERICS AND THE JAVA COLLECTIONS FRAMEWORK

Lecture 15

CS2110 – Spring 2015

Prelim and grading A4

Purpose of prelim: provide feedback on what you learned.
Did you pick up your prelim yet?

We do make mistakes
Did you pick up your prelim yet?

(Gries: I thought I made a mistake once, but I was wrong)

I have ~25 regraded prelims up front. Pick up before/after class if you want. (Regrade notes are on the CMS.)
Then they go in hand-back room Gates 216

Grading A4 starts tomorrow. Will take 5-6 days.

Textbook and Homework

Generics: Appendix B

Generic types we discussed: Chapters 1-3, 15

Useful tutorial:

docs.oracle.com/javase/tutorial/extra/generics/index.html

Testing: A4

Our job: Not to debug our programs but to help you learn how to debug your programs.

Piazza got out of hand with too many emails that were not fruitful.

Sid wrote Piazza note @1033 on testing/debugging. Please read it.

> letters not moving
> Every time I click on the letters, I get a NullPointerException... and they don't move.

A4 took a great deal of time, for many

We will cut back on A5 and attempt to give you good instructions, so it does not take so long.

Not handing it out today because of that.

We should question WHY A4 caused difficulty

Why was A4 so hard (if it was)

- Start early, to have time to ponder, ask questions, get help?
- Did you debug/test properly?
  - BoundingBox methods correct before moving on to BlockTree?
  - Write a Junit test class to test BoundingBox methods? (one purpose of A1 and A3 was to give you practice with this)
  - Put in println statements to help find the cause when your program threw a null-pointer exception or froze or just didn't do the right thing?
  - Execute code by hand, carefully?

BlockTree: one of 2 ArrayLists: 0 blocks

Put println statement in to print info about blocks:

Also, put a return in so that the new BlockTrees would not be constructed, to shorten output.

```
heightbased. lower 0.242…, mid 0.25, high 0.2571…
Block b.position.y: 0.8357142857142857, value: true
Block b.position.y: 0.8357142857142857, value: true
Block b.position.y: 0.8357142857142857, value: true
```

Block not even within box!
Turned out to be mistake in BoundBox.findBox

Time wasted because early method was not correct!

Generic Types in Java

```
/** An instance is a doubly linked list. */
public class LinkedList<E> { ...}
```

You can do this:

```
   LinkedList d= new LinkedList();
   d.append("xy");
```

But this is an error:

```
   String s= d.getFirst().getValue();
```

Need to cast value to String:

```
   String s= (String) d.getFirst().getValue();
```

getValue returns a value of type Object

The cast introduces clutter. It introduces possible runtime errors if wrong cast is done

Generic Types in Java (added in Java 5)

```
/** An instance is a doubly linked list. */
public class LinkedList<E> {
   }
```

Type parameter

You know that in the class, you can use E where ever a type used.

You can do this:

```
LinkedList<Shape> c= new LinkedList<Shape>();
c.append(new Circle(…));
Shape sv= c.getHead().getValue();
```

0. Automatic cast to Shape.

1. No cast is needed, since only Shapes can be appended.
2. Errors caught: illegal to append anything but a Shape to c.
3. Safer, more efficient

IS LinkedList<String> a subtype of LinkedList<Object>?

String is a subclass of Object.

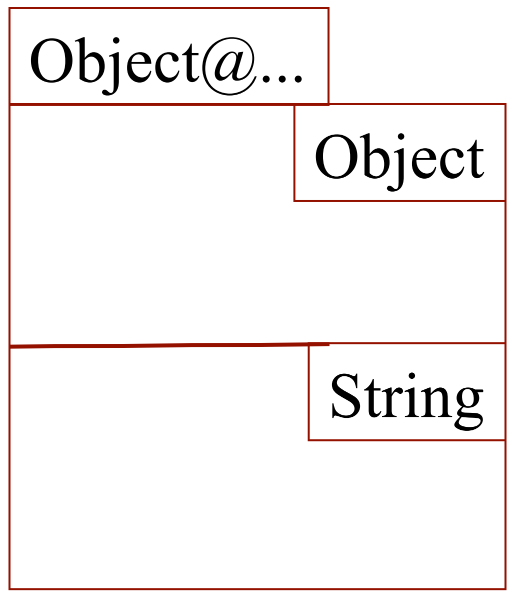
So can store a String in an Object variable:

```
Object ob= "xyx";
```

You might therefore think that

LinkedList<String> is a subtype of

LinkedList<Object>



It is NOT. On the next slide, we explain why it is not ---why allowing that would create an unsafe situation

IS LinkedList<String> a subtype of LinkedList<Object>?

Suppose it is a subtype. Then we can write:

```
LinkedList<String>  ds=   new LinkedList<String>();
LinkedList<Object> do=   ds;  // an automatic upward cast!
do.append(new Integer(55));
```

Linked list ds no longer contains only Strings!

Therefore, Java does not view LL<String> as a subclass of LL<Object>

ds $\boxed{\text{LL<String>@24252424}}$

LL<String>

do $\boxed{\text{LL<String>@24252424}}$

LL<Object>

May be the hardest thing to learn about generics

Suppose S1 is a subclass of S2.

It is not the case that

CL<S1> is a subclass of CL<S2>

Study the previous slide to see why letting CL<S1> be a subclass of CL<S2> would create unsafe situations, ripe for errors

Wild cards: Abbreviate LinkedList by LL

Looks like print, written outside class LL, can be used to print values of any lists

```
/** Print values of ob, one per line. */
public static void print(LL<Object> ob) {
        LL<Object>.Node n= ob.getFirst();
        while (n != null) {
            System.out.println(n.getValue());
            n= n.next();
        }
    }
```

But it won't work on the following because LL<String> is not a subclass of LL<Object>

```
LL<String> d= new LinkedList<String>();
 …
print(d);     // This is illegal
```

Wild cards: Abbreviate LinkedList by LL

Looks like print, written outside class LL, can be used to print any lists' values

```
/** Print values of ob, one per line. */
public void print(LL<Object> ob) {
        LL<Object>.Node n= ob.getFirst();
        while (n != null) {
            System.out.println(n.getValue());
            n= n.next();
        }
}
```

But it won't work on the following because LL<String> is not a subclass of LL<Object>

```
LL<String> d= new LinkedList<String>();
 …
print(d);     // This is illegal
```

Use a wild card ?: Means any type, but unknown

? Is a “wild card”, standing for any type

```
/** Print values of ob, one per line. */
public static void print(LL<?> ob) {
        LL<?>.Node n= ob.getFirst();
        while (n != null) {
              System.out.println(n.getValue());
              n= n.next();
        }
    }
```

It now works!

```
LL<String> d= new LL<String>();
 …
print(d);     // This is legal, because
              // <String> is a class
```

Use a wild card ?: Means any type, but unknown

Looks like print, written outside class LL, can be used to print any lists' values

```
/** Print values of ob, one per line. */
public static void print(LL<?> ob) {
        LL<?>.Node n= ob.getFirst();
        while (n != null) {
            System.out.println(n.getValue());
            ob.append(new Integer(5));
        }
    }
```

But the redline is illegal!
In LL, append's parameter is of type E, and ? Is not necessarily E, so this line is illegal

Bounded wild card

```
/** Print values of ob, one per line. */
public void print(LL<? extends Shape> ob) {
      LL<? extends Shape>.Node n= ob.getHead();
      while (n != null) {
            System.out.println(n.getValue());
            ob.append(new Circle(…)); //Still illegal because type
      }                               // ? Is unknown. Could be Rectangle
   }
```

legal:

```
LL<Circle> dc= …;
print(dc);
```

illegal:

```
LL<JFrame> df= …;
print(df);
```

Can be Shape or any subclass of Shape

Method to append array elements to linked list?

```
/** Append elements of b to d */
public static void m1(Object[] b, LL<Object> d) {
        for (int i= 0; i < b.length; i= i+1 ) {
            d.append(b[i]);
        }
}
```

```
LL<Integer> d= new LL<Integer>();
Integer ia= new Integer[]{3, 5, 6};
m1(ia, d);
```

Doesn't work because:
LL<Integer> not a subtype of LL<Object>

Generic method: a method with a type parameter T

```
/** Append elements of b to d */
public static <T> void m(T[] b, DLL<T> d) {
      for (int i= 0; i < b.length; i= i+1 ) {
          d.append(b[i]);
      }
}
```

type parameter

Don't give an explicit type in the call. Type is inferred.

```
LL<Integer> d= new LL<Integer>();
Integer ia= new Integer[]{3, 5, 6};
m(ia, d);
```

You can have more than one type parameter, e.g. <T1, T2>

Interface Comparable

```
public interface Comparable<T> {
   /** Return a negative number, 0, or positive number
        depending on whether this value is less than, equal to,
        or greater than ob */
   int compareTo(T ob);
}
```

Allows us to write methods to sort/search arrays of any type (i.e. class) provided that the class implements Comparable and thus declares compareTo.

Generic Classes

```
/** = the position of min value of b[h..k].  Pre: h <= k. */
public static  <T>  int  min(Comparable<T>[] b, int h, int k) {
      int p= h;     int i= h;
      // inv: b[p] is the min of b[h..i]
      while (i != k) {
          i= i+1;
          T temp= (T)b[i];
          if (b[p].compareTo(temp) > 0)    p= i;
      }
      return p;
}
```

Java **Collections** Framework

- Collections: holders that let you store and organize objects in useful ways for efficient access

- Package **`java.util`** includes interfaces and classes for a general collection framework

- Goal: conciseness
  - A few concepts that are broadly useful
  - Not an exhaustive set of useful concepts

- The collections framework provides
  - Interfaces (i.e. ADTs)
  - Implementations

JCF Interfaces and Classes

- Interfaces
  - `Collection`
  - `Set (no duplicates)`
  - `SortedSet`
  - `List (duplicates OK)`

  - `Map (i.e. dictionary)`
  - `SortedMap`

  - `Iterator`
  - `Iterable`
  - `ListIterator`

- Classes

  `HashSet`

  `TreeSet`

  `ArrayList`

  `LinkedList`

  `HashMap`

  `TreeMap`

`interface java.util.Collection<E>`

- **public int size();** Return number of elements
- **public boolean isEmpty();** Return true iff collection is empty
- **public boolean add(E x);**
  - Make sure collection includes x; return true if it has changed (some collections allow duplicates, some don' t)
- **public boolean contains(Object x);**
  - Return true iff collection contains x (uses method equals)
- **public boolean remove(Object x);**
  - Remove one instance of x from the collection; return true if collection has changed
- **public Iterator<E> iterator();**
  - Return an Iterator that enumerates elements of collection

Iterators: How “foreach” works

The notation `for(Something var: collection) { … }` is syntactic sugar. It compiles into this “old code”:

```
Iterator<E> _i=  collection.iterator();
while (_i.hasNext()) {
   E var= _i.Next();
     . . . Your code . . .
}
```

The two ways of doing this are identical but the foreach loop is nicer looking.

You can create your own iterable collections

java.util.Iterator<E> (an interface)

public boolean hasNext();

- Return true if the enumeration has more elements

public E next();

- Return the next element of the enumeration
- Throw **NoSuchElementException** if no next element

public void remove();

- Remove most recently returned element by **next()** from the underlying collection
- Throw **IllegalStateException** if **next()** not yet called or if **remove()** already called since last **next()**
- Throw **UnsupportedOperationException** if **remove()** not supported

Additional Methods of `Collection<E>`

public Object[] toArray()

- Return a new array containing all elements of collection

public <T> T[] toArray(T[] dest)

- Return an array containing all elements of this collection; uses dest as that array if it can

Bulk Operations:

- public boolean containsAll(Collection<?> c);
- public boolean addAll(Collection<? extends E> c);
- public boolean removeAll(Collection<?> c);
- public boolean retainAll(Collection<?> c);
- public void clear();

`java.util.Set<E>` (an interface)

- `Set` extends `Collection`
  - `Set` inherits all its methods from `Collection`
- A `Set` contains no duplicates

  If you attempt to `add()` an element twice, the second `add()` will return false (i.e. the `Set` has not changed)

- Write a method that checks if a given word is within a `Set` of words
- Write a method that removes all words longer than 5 letters from a `Set`
- Write methods for the union and intersection of two `Set`s

Set Implementations

java.util.HashSet<E> (a hashtable. Learn about hashing in recitation soon)

- Constructors
  - public HashSet();
  - public HashSet(Collection<? extends E> c);
  - public HashSet(int initialCapacity);
  - public HashSet(int initialCapacity,
                  float loadFactor);

java.util.TreeSet<E> (a balanced BST [red-black tree])

- Constructors
  - public TreeSet();
  - public TreeSet(Collection<? extends E> c);
  - ...

`java.util.SortedSet<E>` (an interface)

- SortedSet *extends* Set

For a SortedSet, the iterator() returns elements in sorted order

- Methods (in addition to those inherited from Set):
  - public E first();
    - Return first (lowest) object in this set
  - public E last();
    - Return last (highest) object in this set
  - public Comparator<? super E> comparator();
    - Return the Comparator being used by this sorted set if there is one; returns null if the natural order is being used
  - …

`java.lang.Comparable<T>` (an interface)

- `public int compareTo(T x);`

  Return a value (< 0), (= 0), or (> 0)
  - (< 0) implies `this` is before `x`
  - (= 0) implies `this.equals(x)`
  - (> 0) implies `this` is after `x`

- Many classes implement `Comparable`
  - `String`, `Double`, `Integer`, `Char`, `java.util.Date`,...
  - If a class implements `Comparable` then that is considered to be the class' s *natural ordering*

`java.util.Comparator<T>` (an interface)

- public int compare(T x1, T x2);
  Return a value (< 0), (= 0), or (> 0)
  - (< 0) implies x1 is before x2
  - (= 0) implies x1.equals(x2)
  - (> 0) implies x1 is after x2

- Can often use a Comparator when a class' s natural order is not the one you want
  - String.CASE_INSENSITIVE_ORDER is a predefined Comparator
  - java.util.Collections.reverseOrder() returns a Comparator that reverses the natural order

SortedSet Implementations

- java.util.TreeSet<E>

  constructors:
  - public TreeSet();
  - public TreeSet(Collection<? extends E> c);
  - public TreeSet(Comparator<? super E> comparator);
  - ...

- Write a method that prints out a SortedSet of words in order
- Write a method that prints out a Set of words in order

`java.util.List<E>` (an interface)

- **List** extends **Collection** items accessed via their index
- Method **add()** puts its parameter at the end of the list
- The **iterator()** returns the elements in list-order
- Methods (in addition to those inherited from **Collection**):
  - **public E get(int i);** Return the item at position i
  - **public E set(int i, E x);** Place x at position i, replacing previous item; return the previous itemvalue
  - **public void add(int i, E x);**
    - Place x at position index, shifting items to make room
  - **public E remove(int index);** Remove item at position i, shifting items to fill the space; Return the removed item
  - **public int indexOf(Object x);**
    - Return index of the first item in the list that equals x (x.equals())
  - …

List Implementations. Each includes methods specific to its class that the other lacks

- java.util.ArrayList<E> (an array; doubles the length each time room is needed)

  Constructors
  - public ArrayList();
  - public ArrayList(int initialCapacity);
  - public ArrayList(Collection<? extends E> c);

- java.util.LinkedList <E> (a doubly-linked list)

  Constructors
  - public LinkedList();
  - public LinkedList(Collection<? extends E> c);

Efficiency Depends on Implementation

- `Object x= list.get(k);`
  - O(1) time for `ArrayList`
  - O(k) time for `LinkedList`
- `list.remove(0);`
  - O(n) time for `ArrayList`
  - O(1) time for `LinkedList`
- `if (set.contains(x)) ...`
  - O(1) expected time for `HashSet`
  - O(log n) for `TreeSet`

What if you need O(1) for both?

- Database systems have this issue

- They often build “secondary index” structures
  - For example, perhaps the data is in an ArrayList
  - But they might build a HashMap as a quick way to find desired items

- The O(n) lookup becomes an O(1) operation!