GENERICS AND THE JAVA COLLECTIONS FRAMEWORK

Lecture 13

CS2110 – Spring 2014

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

Generic Types in Java


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

You can do this:

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

But this is an error:

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

Need to cast value to String:

```
String s= (String) d.getHead().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 DLinkedList<E> {
    }
```

Type parameter

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

You can do this:

```
DLinkedList<Shape> c= new DLinkedList<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

DLinkedList<String> a subtype of DLinkedList<Object>?

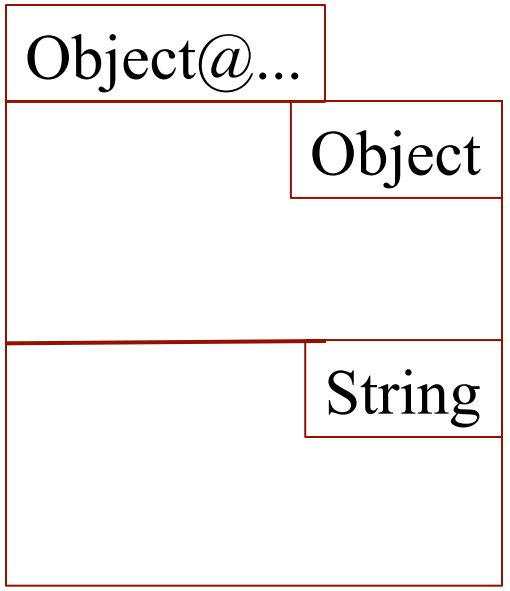
String is a subclass of Object.

So can store a String in an Object variable:

Object ob= “xyx”;

You might therefore think that

DLinkedList<String> is a subtype of DLinkedList<Object>



It is NOT. On the next slide, we explain why it is not!

Is DLinkedList<String> a subtype of DLinkedList<Object>?

Suppose it is a subtype. Then we can write:

```
DLinkedList<String>  ds=   new DLinkedList<String>();
DLinkedList<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 DLL<String> as a subclass of DLL<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> doesn't work.

Wild cards: Abbreviate DLinkedList by DLL

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

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

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

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

Wild cards: Abbreviate DLinkedList by DLL

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

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

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

```
DLL<String> d= new DLinkedList<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 void print(DLL<?> ob) {
        DLL<?>.Node n= ob.getHead();
        while (n != null) {
            System.out.println(n.getValue());
            n= n.successor();
        }
    }
```

It now works!

```
DLL<String> d= new DLL<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 DLL, can be used to print any lists' values

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

But the redline is illegal!

In DLL, 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(DLL<? extends Shape> ob) {
        DLL<? 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:

```
DLL<Circle> dc= …;
print(dc);
```

illegal:

```
DLL<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, DLL<Object> d) {
        for (int i= 0; i < b.length; i= i+1 ) {
            d.append(b[i]);
        }
}
```

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

Doesn't work because:
DLL<Integer> not a subtype of DLL<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.

```
DLL<Integer> d= new DLL<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 **`Sets`**

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!