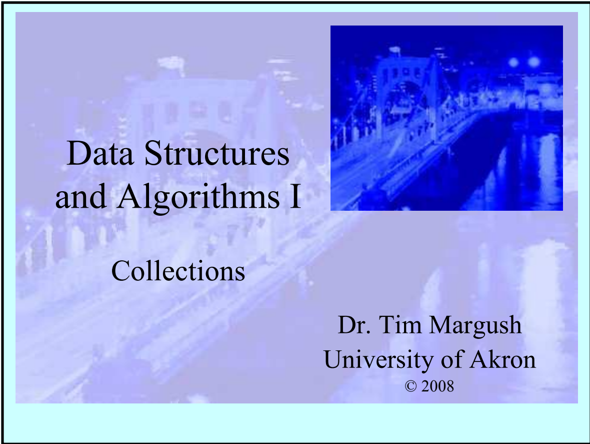
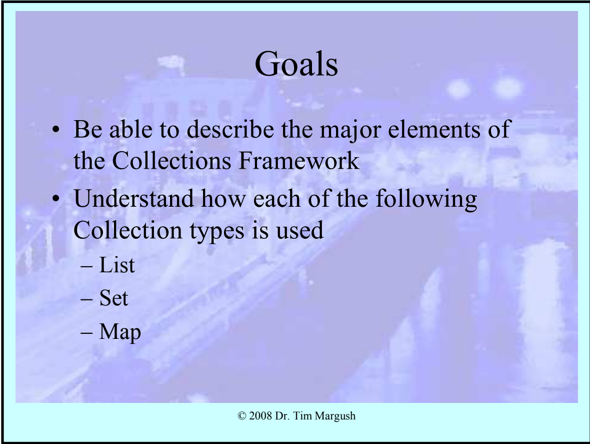
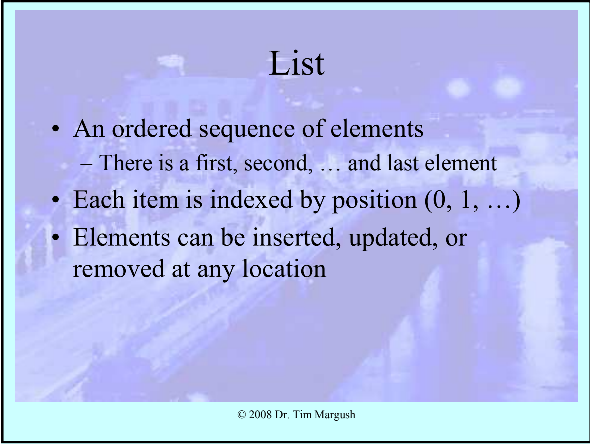

Data Structures
and Algorithms I
Collections
Dr. Tim Margush
University of Akron
© 2008


Goals
• Be able to describe the major elements of the Collections Framework
• Understand how each of the following Collection types is used
– List
– Set
– Map
© 2008 Dr. Tim Margush


List
• An ordered sequence of elements
– There is a first, second, … and last element
• Each item is indexed by position (0, 1, …)
• Elements can be inserted, updated, or removed at any location
© 2008 Dr. Tim Margush

Set

- An unordered collection of elements
  - You do not control the positions of elements
    - Actually, the elements do not have positions
- Duplicate elements are not permitted
- Organized to efficiently detect membership
- Allows insertion and removal (find and remove)

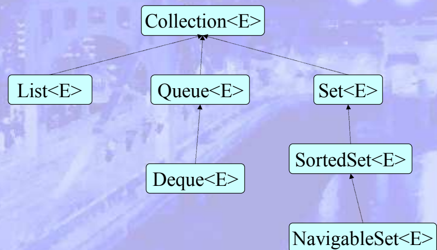
© 2008 Dr. Tim Margush

Map

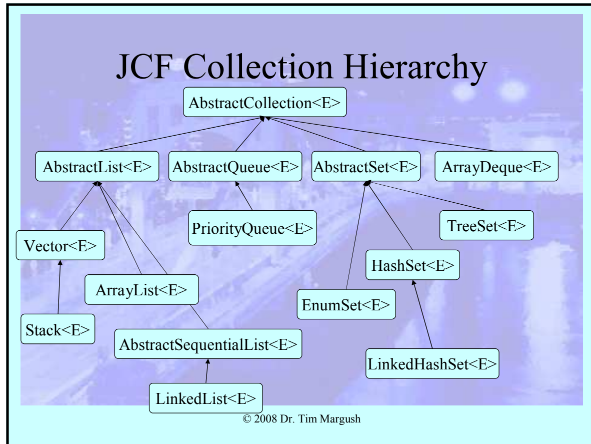
- A set of keys, each connected to a value
  - Considered key, value pairs
- Optimized to locate information by key
- Allows insertion, removal, and update of values based on key

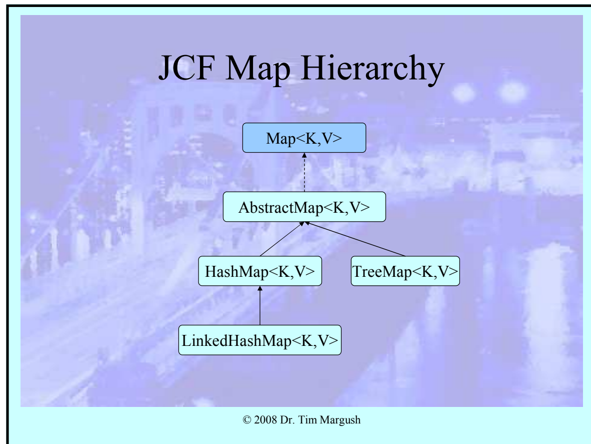
© 2008 Dr. Tim Margush

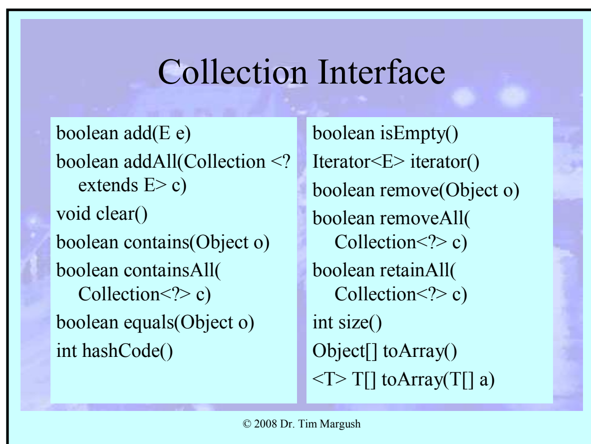
JCF Iterable Interfaces



© 2008 Dr. Tim Margush


JCF Collection Hierarchy
AbstractCollection<E>
AbstractList<E>
AbstractQueue<E>
AbstractSet<E>
ArrayDeque<E>
PriorityQueue<E>
TreeSet<E>
Vector<E>
HashSet<E>
ArrayList<E>
EnumSet<E>
Stack<E>
AbstractSequentialList<E>
LinkedHashSet<E>
LinkedList<E>
© 2008 Dr. Tim Margush


JCF Map Hierarchy
Map<K,V>
AbstractMap<K,V>
HashMap<K,V>
TreeMap<K,V>
LinkedHashMap<K,V>
© 2008 Dr. Tim Margush


Collection Interface
boolean add(E e)
boolean addAll(Collection <? extends E> c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
int hashCode()
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int size()
Object[] toArray()
<T> T[] toArray(T[] a)
© 2008 Dr. Tim Margush

List Interface

```
void add(int index, E element)
boolean addAll(int index, Collection<? extends E> c)
E get(int index)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
E remove(int index)
E set(int index, E element)
List<E> subList(int fromIndex, int toIndex)
```

© 2008 Dr. Tim Margush

AbstractList

- Implements all methods of the interfaces with a default behavior
  - Except get(int p) and size() which must be implemented
  - Backed by a storage structure such as an array
    - You must instantiate the storage structure
- You must override set(int, E), add(E), or remove(int) to modify the list contents

© 2008 Dr. Tim Margush

AbstractList

- Provides a built in iterator
  - Relies on get(int i) and size()
- Calling default methods may throw an OperationNotSupportedException
- removeRange(int from, int to)
  - Gets an iterator positioned before from, and repeatedly calls next and remove (to-from) times
  - This may require $O(n^2)$ time

© 2008 Dr. Tim Margush

Implementation Independent

- Backing structure and method contents may change without affecting the usability
  - May change the efficiency
- Use proper identifiers to access just the methods you need
  - ArrayList implements List
  - ArrayList has ensureCapacity(), List does not
  - Do you need ensureCapacity?
    - List data = new ArrayList();
    - ArrayList data = new ArrayList();

© 2008 Dr. Tim Margush

List of Integers

- int [] list;
- Integer [] list;
- String list;
  - "200000000f220000013c1fffffff4"
    - add(i)
      list+=(Long.toHexString(i+0x200000000L))
    - get(p)
      (int)Long.parseLong(list.substring(p*9,p*9+9), 16)

© 2008 Dr. Tim Margush

Iterators

- List Interface specifies three iterator methods
  - Iterator<E> iterator();
  - ListIterator<E> listIterator();
  - ListIterator<E> listIterator(int index);
    - ListIterator extends Iterator
  - Iterator: hasNext, next, remove
  - ListIterator: add, hasPrevious, previous, nextIndex, previousIndex, set

© 2008 Dr. Tim Margush

Iterator State

- Current position is between two items
  - The one to the right will be returned via next
- hasNext is true if there is an item to the right
- remove must only be called after a next, and only once for each call of next
  - This removes the item just returned by next
- Modifications of the list through other references are not allowed while the iterator is in use
  - ConcurrentModificationException

© 2008 Dr. Tim Margush

Detecting Concurrent Modification

- Lists maintain a modCount variable
  - Incremented for each add or remove
- Iterators check this before list access to ensure it is the expected value
- A simple Iterator can be built using the existing get and size methods

© 2008 Dr. Tim Margush

Simple Iterator

```
class MyIt<E>{
   List<E> theList;
   int pos = 0;
   public MyIt(List<E> t){
        theList = t;
   }
   public E next(){
        return theList.get(pos++);
   }
   public boolean hasNext(){
        return pos < theList.size();
   }
```

© 2008 Dr. Tim Margush

Iterator State

- Current position is between two items
  - The one to the right will be returned via next
  - The one to the left will be returned via previous
  - nextIndex is the index of the one to the right (size of list if at end)
  - previousIndex is the position of the item to the left (-1 if none)
- Add adds between previous and next
- Set requires a previous call to next or previous with no changes to the list in between
  - Replaces the previously returned element with the new one

© 2008 Dr. Tim Margush

Iterators

- The enhanced for loop requires the Iterator interface
- for (E item:list){
  - For each repetition, item is a copy of the next object reference in the collection
- You cannot change the reference in the list
  - although you can modify the object contents if it is mutable
- You cannot use remove; the iterator is hidden from your view

© 2008 Dr. Tim Margush

AbstractSequentialList

- Extends AbstractList by implementing get, add, set, and remove
- However, it requires that you implement a ListIterator and the size method
  - Can you implement the above methods if you have an Iterator? ListIterator?

© 2008 Dr. Tim Margush

Implementing get(int i)

```
public E get(int i){
  Iterator<E> it = iterator();
  while (i > 0) {
      i--; it.next()
  }
  return it.next();
}
```

© 2008 Dr. Tim Margush

Implementing set(int i, E e)

```
public E set(int i, E e){
  Iterator<E> it = iterator();
  while (i > 0) {i--; it.next()};
  E old = it.next();
  it.remove();
  //Now we are stuck… no way to insert
  return old;
}
```

© 2008 Dr. Tim Margush

Implementing set(int i, E e)

```
public E set(int i, E e){
  ListIterator<E> it = listIterator(i);
  E old = it.next();
  it.set(e);
  return old;
}
```

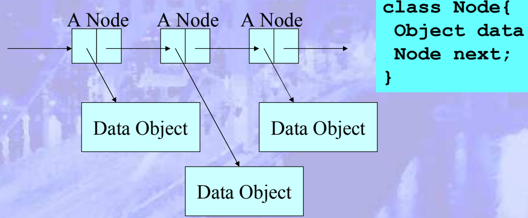
© 2008 Dr. Tim Margush

java.util.LinkedList

- Implements List, but not efficiently (for many methods)
- Optimized for insertion and deletion and sequential access
- Not efficient for random access

© 2008 Dr. Tim Margush

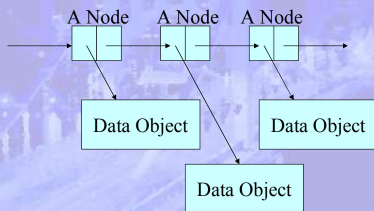
Linked List Concept



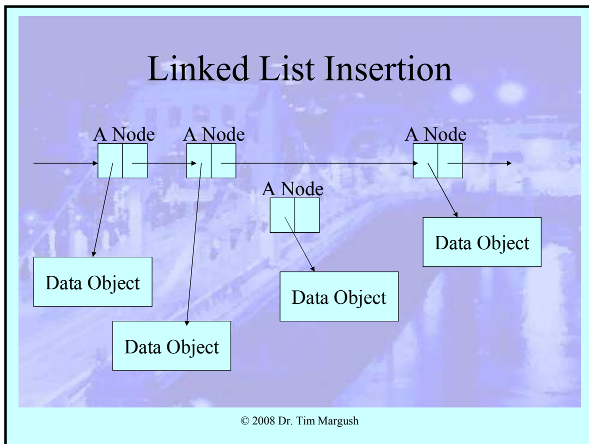

```
class Node{
 Object data
 Node next;
}
```

© 2008 Dr. Tim Margush

Linked List Removal



© 2008 Dr. Tim Margush



Hash Set

- HashSet<String> hs =
  new HashSet<String>();
- hs.add("Fred");
  - Does not permit duplicates
- if (hs.contains("Tommy"))…
  - Uses .hashCode and .equals
  - Is O(1) search
- for (String n : hs) …

© 2008 Dr. Tim Margush

Hash Map

- HashMap<String, Student> hm =
  new HashMap<String, Student>();
- hm.put("22-3456", aStudent);
  - Reusing a key will overwrite previous Student value at t at location
- Student s = hm.get("32-0983");
  - Uses .hashCode and .equals of the key
  - Is O(1) search
- for (String skey : hm.keySet()) …

© 2008 Dr. Tim Margush

Collections Class

- Static methods that process collections

```
public static int binarySearch(
   List<? extends Comparable<? super T>> list,
   T key);
public static <T> void copy(
   List<? super T> dest, List<? extends T> src);
public static <T> List<T> nCopies(
   int n, T o);
public static void shuffle(List<?> list);
publid static <T extends Comparable<? super T>>
  void sort(List<T> sort);
```

© 2008 Dr. Tim Margush

Summary

- The Java Collection and Map Frameworks provide
  - Interfaces
  - Abstract Implementations
  - Concrete Classes
- Each is tailored to specific types of problems
- Subclassing is commonly used to enforce custom functionality

© 2008 Dr. Tim Margush