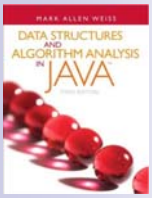
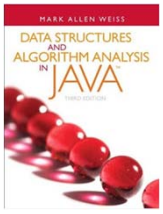


FIU FLORIDA INTERNATIONAL UNIVERSITY

Lists, Stacks, and Queues (II)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences



COP-3530 - Data Structures

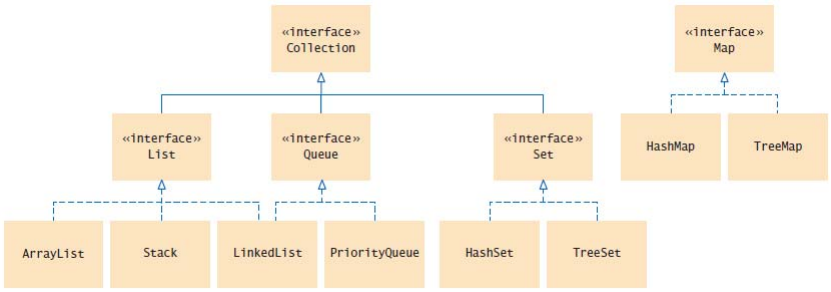
Module #2: Lists, Stacks, and Queues (part II)

Outline:

- **List in the Java collections API.**

Lists in the Java Collections API

- **Collections API** – set of **classes** and **interfaces** in Java that implement commonly reusable collection data structures.
- **Collections API** resides in package **java.util**.



3

Lists in the Java Collections API

- Example of the most important parts of this interface:

```
public interface Collection<AnyType> extends Iterable<AnyType>
{
        int size( );
        boolean isEmpty( );
        void clear( );
        boolean contains( AnyType x );
        boolean add( AnyType x );
        boolean remove( AnyType x );
        java.util.Iterator<AnyType> iterator( );
}
```

4

Lists in the Java Collections API. Iterators

- The **Collection** interface extends the **Iterable** interface.
- Classes that implement the **Iterable interface** can have the **enhanced for loop** used on them to view all their items.

```
public static <Any Type> void print( Collection<AnyType> coll )
{
    for ( AnyType item : coll )
        System.out.println( item );
}
```

5

Lists in the Java Collections API. Iterators

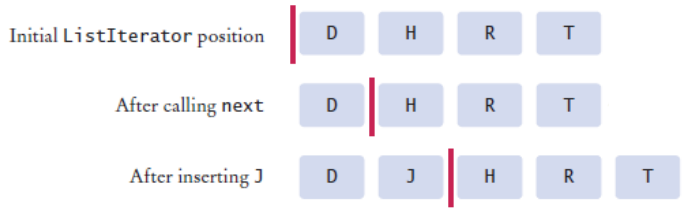
```
public interface Iterator<AnyType>
{
    boolean hasNext( );
    AnyType next( );
    void remove( );
}
```

- **Main Idea:**
  - Via the **iterator method**, each collection can create, and return an object that implements the Iterator interface and stores internally its notion of a current position.

6

Lists in the Java Collections API. Iterators

- The Collections that implement the **Iterable interface** must provide a method named **iterator** that returns an object of type Iterator.
- The **iterator** is an interface defined in **java.util**
- Think of an **iterator** as pointing between two elements:
  - Analogy: like the cursor in a word processor points between two characters



7

Lists in the Java Collections API. Iterators

```
public static <AnyType> void print( Collection<AnyType> coll )
  {
       for( AnyType item : coll )
            System.out.println( item );
  }
```

- When the compiler sees an **enhanced for loop** being used on an object that is **Iterable**, it mechanically replaces the enhanced for loop with calls to the iterator method to obtain an Iterator and then calls to next and **hasNext**.
- Thus the previously "print routine" is rewritten by the compiler as:

```
public static <AnyType> void print( Collection<AnyType> coll )
{
        Iterator<AnyType> itr = coll.iterator( );
        while( itr.hasNext( ) )
        {
            AnyType item = itr.next( );
            System.out.println( item );
        }
}
```

8

List interface in java.util

- Subset of the **List interface** in package **java.util**

```
public interface List<AnyType> extends Collection<AnyType>

{
    AnyType get( int idx );
    AnyType set( int idx, AnyType newVal );
    void add( int idx, AnyType x );
    void remove( int idx );
    ListIterator<AnyType> listIterator( int pos );
}
```

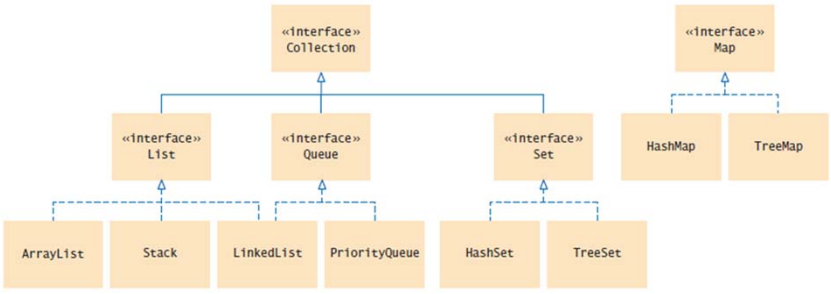
- **get** and **set** - to access or change an item at the specified position in the list, given by its index, idx.
  - Index 0 - the front of the list,
  - Index size()-1 - the last item in the list,
  - Index size() - the position where a newly added item can be placed.
- **add** – to add a new item in position idx (pushing subsequent items one position higher).
  - add at position 0 – add at the front,
  - add at position size() is adding an item as the new last item

9

List interface in java.util

- Two popular implementations of the **List ADT**.

ArrayList and **LinkedList**



10

List interface in java.util

- (+) Advantage of **ArrayList**
  - **get** and **set** take constant time.
- (-) Disadvantage of **ArrayList**
  - **insertion** of new items and **removal** of existing items is expensive
- (+) Advantage of **LinkedList** (doubly linked list)
  - **insertion** of new items and removal of existing items is cheap.
  - adds and removes from the front of the list are constant-time operations.
  - LinkedList has methods **addFirst**, **removeFirst**, **addLast**, **removeLast**, **getFirst** and **getLast** to efficiently add, remove, and access the items at both ends of the list.
- (-) Disadvantage of **LinkedList**
  - **is not easily indexable**, so calls to **get** are expensive

11

List interface in java.util

- Construct a **List by adding items at the end**.

```
public static void makeList1( List<Integer> lst, int N )
{
        lst.clear( );
        for( int i = 0; i < N; i++ )
            lst.add( i );
}
```

  - For **ArrayList** or **LinkedList** the running time of **makeList1** is $O(N)$ because each call to add, being at the end of the list is $O(1)$ time (the occasional expansion of the ArrayList is safe to ignore).
- Construct a **List by adding items at the front**.

```
public static void makeList2( List<Integer> lst, int N )
{
        lst.clear( );
        for( int i = 0; i < N; i++ )
            lst.add( 0, i );
}
```

  - The running time is $O(N)$ for a **LinkedList**, but $O(N^2)$ for an **ArrayList**, because in an **ArrayList**, adding at the front is an $O(N)$ operation.

12

List interface in java.util

- Compute the **sum of the numbers in a List**:

```
public static int sum( List<Integer> lst )
{
      int total = 0;
      for( int i = 0; i < N; i++ )
         total += lst.get( i );
      return total;
}
```

  - **sum** is $O(N)$ for an **ArrayList**, but $O(N^2)$ for a **LinkedList**, because in a **LinkedList**, calls to get are $O(N)$ operations.

DoublyLinkedList.java

- See Example in:

  http://algs4.cs.princeton.edu/13stacks/DoublyLinkedList.java.html


15