CS3500: Object-Oriented Design
Fall 2013

Class 9
10.4.2013
Readings

• Sestoft Section 13: Interfaces

• Sestoft Section 22.7: Going Through a Collection: Interfaces Iterator<T> and Iterable<T>
Iterators
Iterators

[Lewis & Chase]

• An *iterator* is an object that provides the means to iterate over a collection.

• Provide methods that allow the user to acquire and use each element in a collection in turn.
/An iterator over a collection.
public interface Iterator<E> {
    // Returns true if the iteration has more
    // elements.
    public boolean hasNext ();

    // Returns the next element in the iteration.
    public E next ();

    // Removes from the underlying collection the last
    // element returned by the iterator (optional).
    public void remove ();
}
Iteration Abstraction

[Liskov, p. 130]

• An *iterator* is a procedure that returns a *generator*. A data abstraction can have one or more iterator methods, and there can also be standalone iterators.

• A generator is an object that produces the elements used in the iteration. It has methods to get the next element and to determine where there are any more elements. The generator’s type is a subtype of `Iterator`.

• The specification of an iterator defines the behavior of the generator; a generator has no specification of its own. The iterator specification often includes a requires clause at the end constraining the code that uses the generator.
ArrayListIterator
Nested Classes
Nested Classes

[http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html]

class OuterClass {
    
    ... 

    class NestedClass {
        
        ... 

    }

}
Nested Classes

[http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html]

class OuterClass {

    ...

    static class StaticNestedClass {

        ...

    }

    class InnerClass {

        ...

    }

}
Why Use Nested Classes?

[http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html]

• It is a way of logically grouping classes that are only used in one place

• It increases encapsulation

• It can lead to more readable and maintainable code
ArrayListIterator
Nested Classes

class ArrayListIterator implements Iterable<String> {
    
    class AListIterator implements Iterator<String> {
        
    }
}

}
Refactoring with Nested Classes
Binary Search
Binary Search

If a set $S$ is represented by a sorted linear sequence, then we can determine whether $x$ is an element of $S$ in logarithmic time by using binary search.
Binary Search

Search for 51
Binary Search

Search for 51
Binary Search

Search for 51
Binary Search

Search for 51
Binary Search

Start

Search for 51

FOUND
Binary Search

Search for 53
Binary Search

```
1  3  7  11 19 22 25 33 41 49 51 57 60
```

Search for 53
Binary Search

Search for 53
Binary Search

Search for 53
Search for 53

NOT FOUND
/**
* PRE: min <= max
* PRE: 0 <= min, max <= data.length
* @param data sorted array of ints
* @param min min index to search
* @param max max index to search
* @param target value searching for in data
* @return index of target in data,
*          or -1 if target is not in data
*/
public static int binarySearch(int[] data, int min, int max,
       int target){

}
/**
 * PRE: min <= max
 * PRE: 0 <= min, max <= data.length
 * @param data sorted array of ints
 * @param min min index to search
 * @param max max index to search
 * @param target value searching for in data
 * @return index of target in data,
 *         or -1 if target is not in data
 */

public static int binarySearch(int[] data, int min, int max, int target) {
    int index = -1; // not in array
    int midpoint = (min + max) / 2;

    if (data[midpoint] == target) {
        index = midpoint;
    } else {
        if (data[midpoint] > target) {
            if (min <= midpoint - 1) {
                index = binarySearch(data, min, midpoint - 1, target);
            } else {
                if (midpoint + 1 <= max) {
                    index = binarySearch(data, midpoint + 1, max, target);
                }
            }
        }
    }
    return index;
}
Total Order
Total Order

A total order on some set $D$ is a binary relation $R$ on $D$ such that

- $R$ is transitive
- $R$ is anti-symmetric
- $R$ satisfies the law of trichotomy
Total Order

A *total order* on some set $D$ is a binary relation $R$ on $D$ such that

- $R$ is **transitive**: if $xRy$ and $yRz$, then $xRz$
- $R$ is anti-symmetric
- $R$ satisfies the law of trichotomy
Total Order

A *total order* on some set $D$ is a binary relation $R$ on $D$ such that

- $R$ is transitive
- $R$ is **anti-symmetric**: if $xRy$ and $yRx$, then $x = y$
- $R$ satisfies the law of trichotomy
Total Order

A total order on some set $D$ is a binary relation $R$ on $D$ such that

- $R$ is transitive
- $R$ is anti-symmetric
- $R$ satisfies the law of trichotomy: $\forall x, \forall y$ either $xRy$ or $yRx$

The law of trichotomy (a division into three categories) can also be phrased as $\forall x, \forall y$ either

- $x=y$
- $x\neq y$ and $xRy$
- $x\neq y$ and $yRx$
Examples of Total Orders
Usual Ordering on Integers

\((R \leq)
\)

- \(R\) is transitive
- \(R\) is anti-symmetric
- \(R\) satisfies the law of trichotomy
Reverse Ordering on Integers

$(R :>=)$

- $R$ is transitive
- $R$ is anti-symmetric
- $R$ satisfies the law of trichotomy
Ordering on Integers

- every even integer is less than every odd integer
- the even integers are ordered by the usual \( \leq \)
- the odd integers are ordered by the reverse \( \geq \)

- \( R \) is transitive
- \( R \) is anti-symmetric
- \( R \) satisfies the law of trichotomy
Trees
Binary Trees
Labeled Binary Tree (LBT)

- an empty tree
- a node with three components:
  - a label
  - a left subtree, which is a labeled binary tree
  - a right subtree, which is a labeled binary tree
Binary Search Trees
Binary Search Tree (BST)

- $t$ is empty
- $t$ is a node
  - a label
  - the left subtree of $t$ is a BST,
  - the right subtree of $t$ is a BST,
  - every label within the left subtree of $t$ is less than the label of $t$,
  - every label within the right subtree of $t$ is greater than the label of $t$
Tree Traversals

- We’d like to visit each data item in a tree

- Are the items randomly ordered, as in a bag or set?

- Think of visiting the data in a node, and its left and right subtrees, in some order
Preorder Traversal

Order of nodes visited:
D

Visit the data
Visit the left subtree
Visit the right subtree
Preorder Traversal

Order of nodes visited:

D  B

Visit the data
Visit the left subtree
Visit the right subtree
Preorder Traversal

Order of nodes visited:
D  B  A

Visit the data
Visit the left subtree
Visit the right subtree
Preorder Traversal

Order of nodes visited:

D  B  A  C

Visit the data
Visit the left subtree
Visit the right subtree
Preorder Traversal

Order of nodes visited:
D  B  A  C  F

Visit the data
Visit the left subtree
Visit the right subtree
Preorder Traversal

Order of nodes visited:
D B A C F E

Visit the data
Visit the left subtree
Visit the right subtree
Preorder Traversal

Order of nodes visited:

D B A C F E G

Visit the data
Visit the left subtree
Visit the right subtree
Inorder Traversal

Order of nodes visited:

A

Visit the left subtree
Visit the data
Visit the right subtree
Inorder Traversal

Order of nodes visited:

A B

Visit the left subtree
Visit the data
Visit the right subtree
Inorder Traversal

Order of nodes visited:

A B C

Visit the left subtree
Visit the data
Visit the right subtree
Inorder Traversal

Order of nodes visited:
A B C D

Visit the left subtree
Visit the data
Visit the right subtree
Inorder Traversal

Order of nodes visited:
A B C D E

Visit the left subtree
Visit the data
Visit the right subtree
Inorder Traversal

Order of nodes visited:
A B C D E F

Visit the left subtree
Visit the data
Visit the right subtree
Inorder Traversal

Order of nodes visited:

A B C D E F G

Visit the left subtree
Visit the data
Visit the right subtree
Postorder Traversal

Order of nodes visited:

Visit the left subtree
Visit the right subtree
Visit the data
Postorder Traversal

Order of nodes visited:

A C

Visit the left subtree
Visit the right subtree
Visit the data
Postorder Traversal

Order of nodes visited:
A C B

Visit the left subtree
Visit the right subtree
Visit the data
Postorder Traversal

Order of nodes visited:
A C B E

Visit the left subtree
Visit the right subtree
Visit the data
Postorder Traversal

Order of nodes visited:

A C B E G

Visit the left subtree
Visit the right subtree
Visit the data
Postorder Traversal

Order of nodes visited:
A C B E G F

Visit the left subtree
Visit the right subtree
Visit the data
Postorder Traversal

Order of nodes visited:

A C B E G F D

Visit the left subtree
Visit the right subtree
Visit the data
Comparator<T>