CSE 12 – Basic Data Structures and Object-Oriented Design
Lecture 24

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

Announcements

- Quiz 24 due Wednesday @ 8am
- Survey 10 due Friday @ 11:59pm
- PA8 due Thursday @ 11:59pm
  - No resubmission
- PA7 Resubmission due Friday @ 11:59pm
- Final Exam
  - Starts Saturday, March 13th @ 8:00am
  - Ends Monday, March 15th @ 11:59pm
  - 3 hour exam – clock starts when you open exam
    - Must be finished in one sitting

Topics

- Iterators
- Questions on Lecture 24?

Iterable Interface

Implementing this interface allows an
object to be the target of the "for-each loop" statement.

```
// Defined in java.lang
public interface Iterable <T> {
  /** Returns an iterator over elements of type T. */
  Iterator<T> iterator();

}
```

Iterable Interface

```
// Defined in java.lang
public interface Iterable <T> {
  /** Returns an iterator over elements of type T. */
  Iterator<T> iterator();

}
```

This interface is mostly used when we create a container of certain things and we want to be able to iterate through it without worrying about indexes.

- There is no sorting involved.
- You can use the iterator from other Java containers to get the iterator

```
import java.util.Arrays;
class Person{
  private String name;
  public Person(){ name = null;}
  public Person( String name ){ this.name = name;}
  public String toString(){ return name; }
}
class CSE12 implements Iterable<Person>{
  private Person[] data;
  public CSE12 (){
    data = new Person[3];
    data[0] = new Person("Greg");
    data[1] = new Person("Paul");
    data[2] = new Person("Christine");
  }
  public Iterator<Person> iterator(){
    return Arrays.asList(data).iterator();
  }}
public class IterableTest{
  public static void main(String[] args){
    CSE12 obj = new CSE12();
    for (Person p : obj){
        System.out.println(p);
    }}}
```

When this code runs, what will be printed out?

A. Christine
Greg
Paul

B. Gerald
Paul
Christine

C. Some random order

The Iterator Software Design Pattern

- A common situation: A client needs to inspect the data elements in a collection, without wanting to know details of how the collection structures its data internally

- Solution:
  - Define an interface that specifies how an iterator will behave
  - Design the collection to be able to supply an object that implements that iterator interface
  - A client then can ask the collection for an iterator object, and use that iterator to inspect the collection's elements, without having to know how the collection is implemented

`Iterator<E>` Interface

https://docs.oracle.com/javase/10/docs/api/java/util/Iterator.html

The `Iterator<E>` interface is defined as follows:

```
public interface Iterator<E> {
    default void forEachRemaining(Consumer<? super E> action)
    public E next();
    public boolean hasNext();
    default public void remove();
  }
```

- The ListIterator<E> interface extends the Iterator<E> interface and adds a few more methods…

public interface ListIterator<E> extends Iterator<E>

https://docs.oracle.com/javase/10/docs/api/java/util/ListIterator.html

boolean hasNext()

Return true if there are more elements when going in the forward direction.

T next()

Return the next element in the list when going forward.

Throw NoSuchElementException if there is no such element

boolean hasPrevious()

Return true if there are more elements when going in the reverse direction.

T previous()

Return the next element in the list when going backwards.

Throw NoSuchElementException if there is no such element

next

`E next()`

Returns the next element in the list and advances the cursor position. This method may be called repeatedly to iterate through the list, or intermixed with calls to `previous()` to go back and forth. (Note that alternating calls to `next` and `previous` will return the same element repeatedly.)

Specified by:

`next` in interface `Iterator<E>`

Returns:

the next element in the list

Throws:

`NoSuchElementException` - if the iteration has no next element

List Traversal

```
LinkedList<Integer> myL = new LinkedList<Integer>();
// Add the elements 1, 2, 3, 4 to myL (code not shown)
ListIterator<MyObjs> it = myL.listIterator();
while (it.hasNext()) {
     System.out.println(it.next());
}
```

```
LinkedList<Integer> myL = new LinkedList<Integer>();
// Add the elements 1, 2, 3, 4 to myL (code not shown)

for (int i = 0; i<myL.size(); i++) {
     System.out.println(myL.get(i));
}
```

Is there a difference in the behavior between these two blocks of code?

A. Yes
B. No
C. Yes, but not in a way that the user would be able to tell

Random Stream

- How could we make our random stream so that we can get random numbers in an enhanced for loop?
- Random random;
- **public int nextInt(int bound)**
  - Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
- public RandomStream(int size, int bound) { }

```
RandomStream r = new RandomStream(10, 100);
for (Integer i : r) {
  System.out.println(i);
}
```

Other Iterators

- How would we make a BST iterator?
- What about a Heap iterator?

Questions on Lecture 24?