Iterator

COMP 401, Spring 2013

Lecture 07

1/31/2013

Design Situation

- Suppose we have an object that encapsulates some sort of collection.
  - SongLibrary
    - A collection of songs in an iTunes-like system
  - PolygonModel
    - A collection of polygons in a 3D modeling system
  - Polygon
    - A collection of points in our Assignment 2

Design Situation

- Now suppose we have code *outside* of this collection object that needs to examine each element of the underlying collection in turn.
  - SongFilter
    - A object that represents a set of search criteria that is to be applied to a collection of songs
  - An intersection test in which each polygon of a PolygonModel needs to be evaluated

Strategy 1: Provide access to underlying collection as an array.

- SongLibrary
  - public Song[] getSongs()
- PolygonModel
  - public Polygon[] getPolygons()
- Drawbacks?
  - May have to do a lot of work to create the array
  - Collection may be result of generative process

Strategy 2: Provide index access to each underlying item in collection

- SongLibrary
  - public int getNumSongs();
  - public Song getSong(int song_idx);
- PolygonModel
  - public int getNumPolygons();
  - public Polygon getPolygon(int polygon_idx);
- Drawbacks?
  - Doesn't help with generative collections
  - Imposes restrictions on how collection is represented and linearized
  - Deteriorates encapsulation

Strategy 3: Internalize a "cursor"

- SongLibrary
  - public void resetSongCursor();
  - public Song getNextSong();
  - public boolean isCursorAtEnd();
- Drawbacks?
  - Can't have two traversals going at the same time.
  - But, this does come close.

Iterator Design Pattern

- "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"
  - Gang of Four, *Design Patterns*
- Consider:

```
for(int i=0; i<slist.size(); i++) {
   Song next_song = slist.get(i);
   // Do something with next_song.
}
```

Iterator Design Pattern

- Iterator object encapsulates details of item traversal.
  - Understands details of the underlying collection.
  - Manages order of items
    - May want a traversal that is not just first to last.
    - Underlying collection may not be linear.
  - Manages state of traversal
    - Allows traversal to be picked up again later.
- Assumption: underlying collection is not changed or modified while the traversal is occurring.
  - Interator should be able to detect this and signal an error
  - Variant of pattern will have iterator provide methods that modify underlying collection safely

Elements of Iterator Pattern

- Collection object is “iterable”
  - Provides a method that returns an object that acts as an iterator.
- Iterator object provides access to the elements in turn.
  - A method to test whether more items exist.
  - A method to retrieve the next item.

Java Iterator Pattern Interfaces

- The Java Collections Framework defines two generic interfaces for supporting the interable design pattern
  - Implemented by the various collection types such as List<E>, Map<E>, Set<E>, etc.
- Iterable<E>
  - Interator<E> iterator()
- Iterator<E>

Iterator<E>

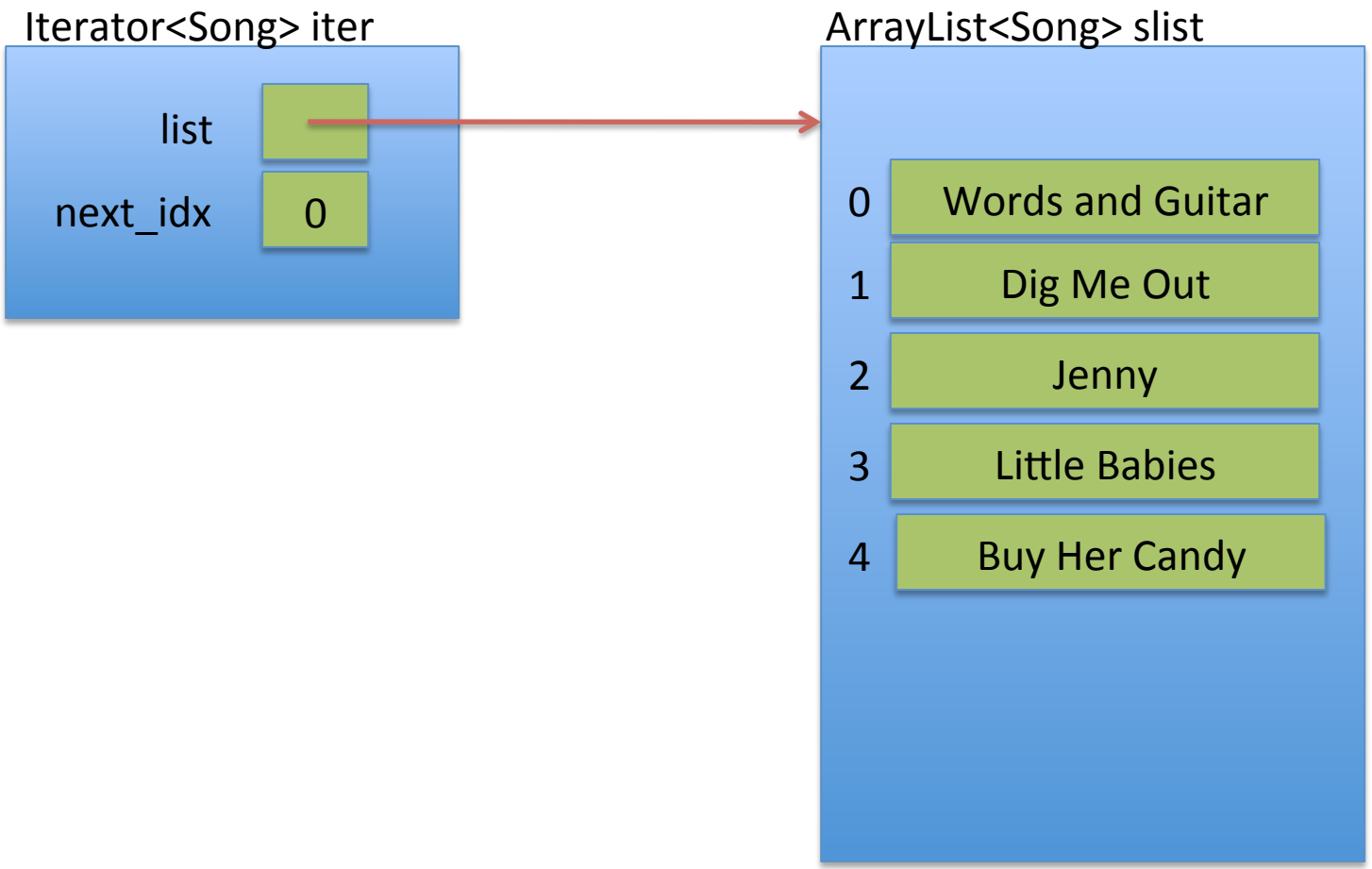
- boolean hasNext()
  - Are we at the end of the traversal?
- E next()
  - Get the next item of the traversal.
  - Throws a runtime exception if no next item.
- void remove()
  - Not supported by all implementations.
  - Safely removes last item retrieved by next() from the underlying collection.

Iterable examples

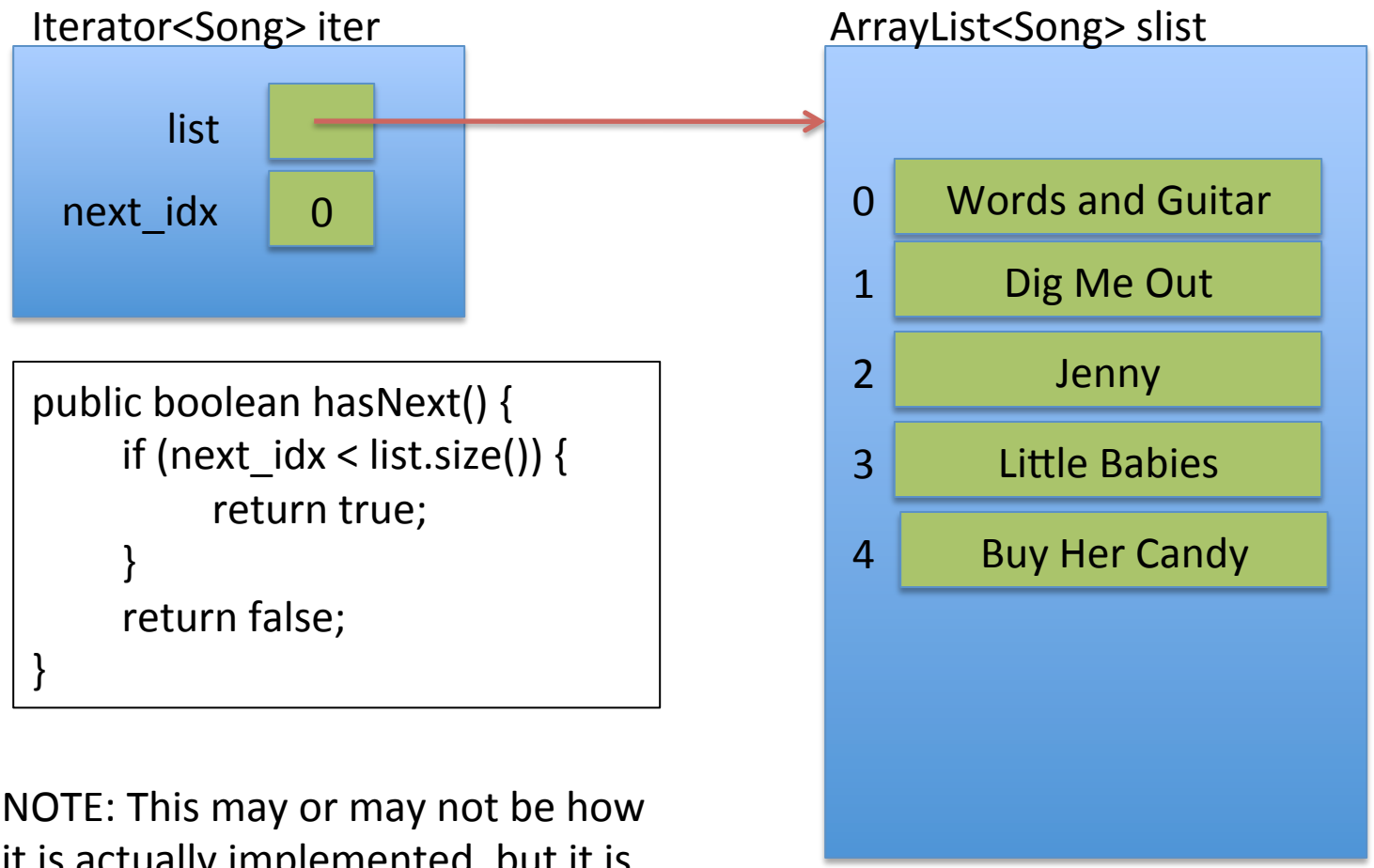
- lec7.v1
  - Main1
    - Simple use
  - Main2
    - Parallel iterators
  - Main3
    - Simultaneous iterators
  - Main4
    - for – each syntactic sugar

Main1 Visualized (1)

Main1 Visualized (2)



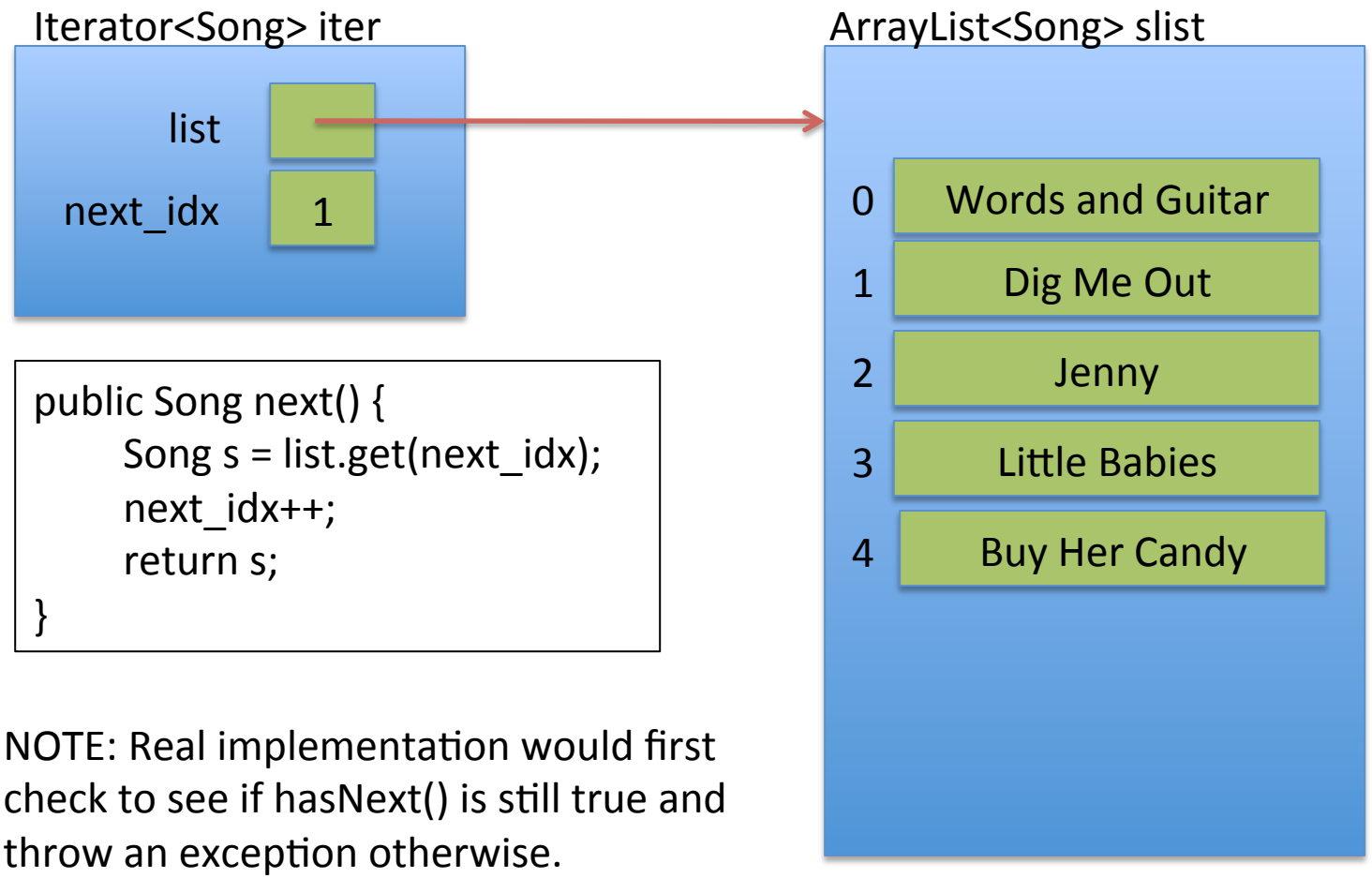
Main1 Visualized (3)




```
public boolean hasNext() {
        if (next_idx < list.size()) {
                return true;
        }
        return false;
}
```

NOTE: This may or may not be how it is actually implemented, but it is *effectively* what is going on.

Main1 Visualized (4)




```
public Song next() {
        Song s = list.get(next_idx);
        next_idx++;
        return s;
}
```

NOTE: Real implementation would first check to see if hasNext() is still true and throw an exception otherwise.

lec7.v1.Main2

- Parallel iteration
  - Processes two different lists
    - Iterator associated with each.
    - Iterators advance unevenly

lec7.v1.Main3

- Simultaneous iteration
  - 2 Iterators, 1 List
    - Insert your own joke here.

lec7.v1.Main4

- Java provides “syntactic sugar” for common use of iterators.
  - Supposing e_coll is Iterable<E>, then these are equivalent:

```
Iterator<E> iter = e_coll.iterator();
while (iter.hasNext()) {
    E elem = iter.next();
    // Do something with element
}
```

```
for (E elem : e_coll) {
    // Do something with elem
}
```

lec7.v2

- A more complicated iterator
  - Can build iterators that do things other than just go through every item.
    - Prior examples made use of Iterator<E> built into List<E>, here we are going to implement our own specialized version of Iterator<E>