Java Collections

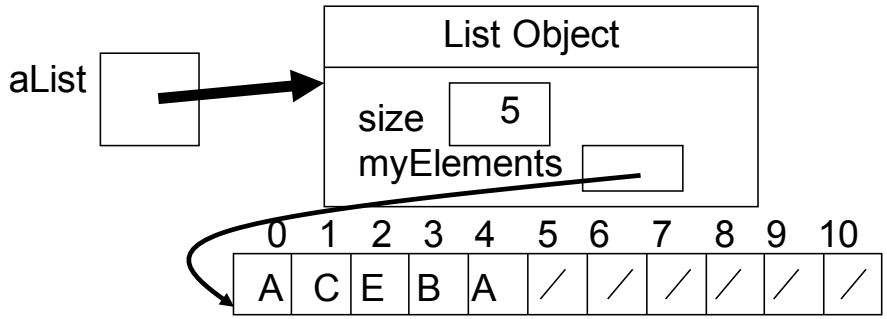
Ye Yang
Stevens Institute of Technology

Readings and References

- Textbook: Chapter 6
- Online references: "Collections", Java tutorial
  - http://java.sun.com/docs/books/tutorial/collections/index.html

Data Structures

- A *Data Structure*
  - a representation of data and the operations allowed on that data
- Allows to achieve important OOP goal
  - Component reuse
  - Better algorithm efficiency
- Part of the Java Standard Library is the *Collections Framework*
  - Most resides in *java.util*
  - A collection of data structures
  - Built on two interfaces
    - Collection
    - Iterator



High Level Protocol

- Data Structures will have 3 core operations
  - a way to add things
  - a way to remove things
  - a way to access things
- Details of these operations depend on the data structure
  - Example: List, add at the end, access by location, remove by location
- More operations added depending on what data structure is designed to do

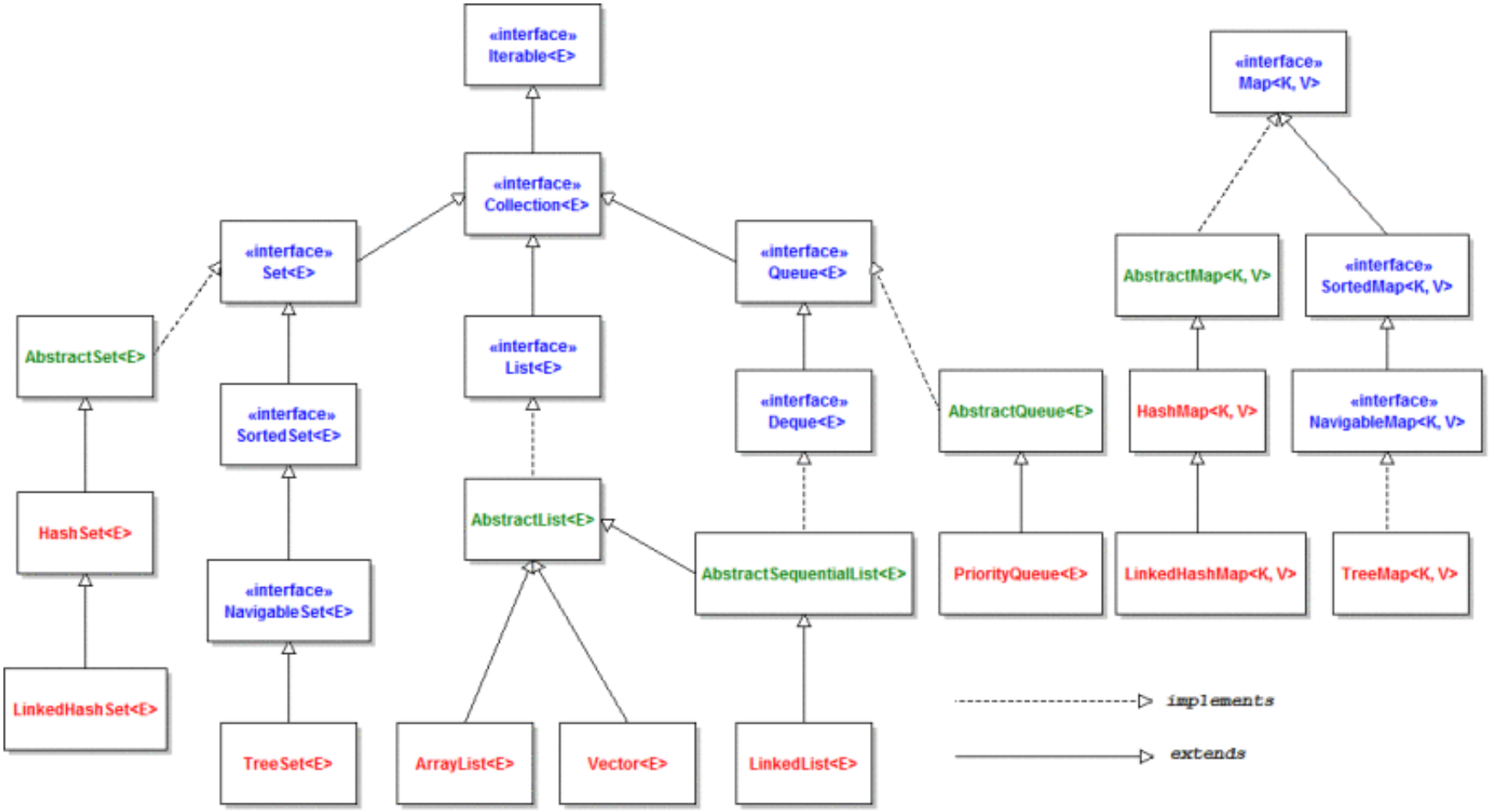
Need for Collection Framework

```
// Java program to show why collection framework was needed
import java.io.*;
import java.util.*;

class CollectionDemo{
        public static void main (String[] args){
                // Creating instances of array, vector and hashtable
                int arr[] = new int[] {1, 2, 3, 4};
                Vector<Integer> v = new Vector();
                Hashtable<Integer, String> h = new Hashtable();

                v.addElement(1);        // Vector element insertion requires addElement()
                v.addElement(2);
                h.put(1,"geeks");       // hashtable element insertion requires put()
                h.put(2,"4geeks");

                // Accessing first element of array, vector and hashtable
                System.out.println(arr[0]);
                System.out.println(v.elementAt(0));
                System.out.println(h.get(1));
        }
}
```

Output:

```
1
1
geek
```

Collections Framework Diagram



- Interfaces, Implementations, and Algorithms

Advantages of Collection Framework

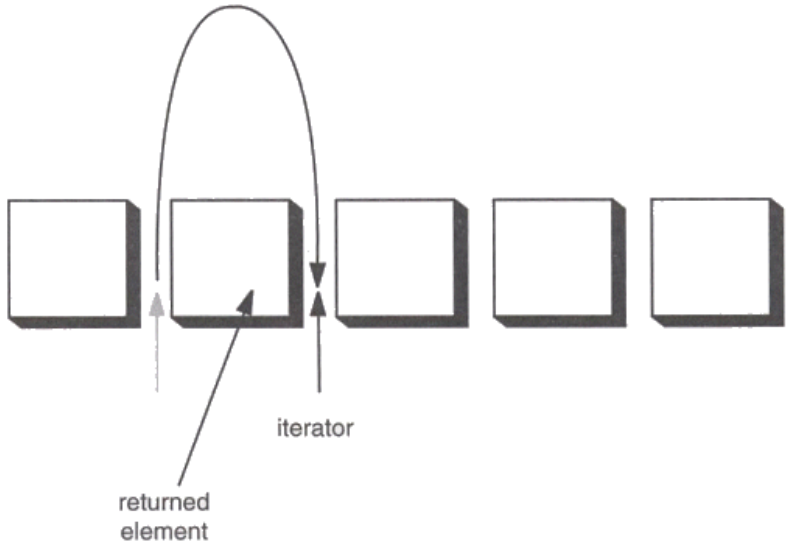
- Consistent API
  - The API has basic set of interfaces like **Collection, Set, List, or Map.**
  - All classes (such as **ArrayList, LinkedList, Vector** etc.) which implements these interfaces have some common set of methods.
- Reduces effort to learn and to use new API
  - The programmers need not to worry about design of Collection rather than he can focus on its best use in his program.
  - Fosters software reuse
- Increases program speed and quality
  - Increases performance by providing high-performance implementations of useful data structures and algorithms.

Collection Interface

- Defines fundamental methods
  - `int size();`
  - `boolean isEmpty();`
  - `boolean contains(Object element);`
  - `boolean add(Object element);`
  - `boolean remove(Object element);`
  - `Iterator iterator();`
- These methods are enough to define the basic behaviors of a collection
- Provides an Iterator to step through the elements in the Collection

Iterator Interface

- Defines three fundamental methods
  - `Object next()`
  - `boolean hasNext()`
  - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
  - Then you can use it or remove it

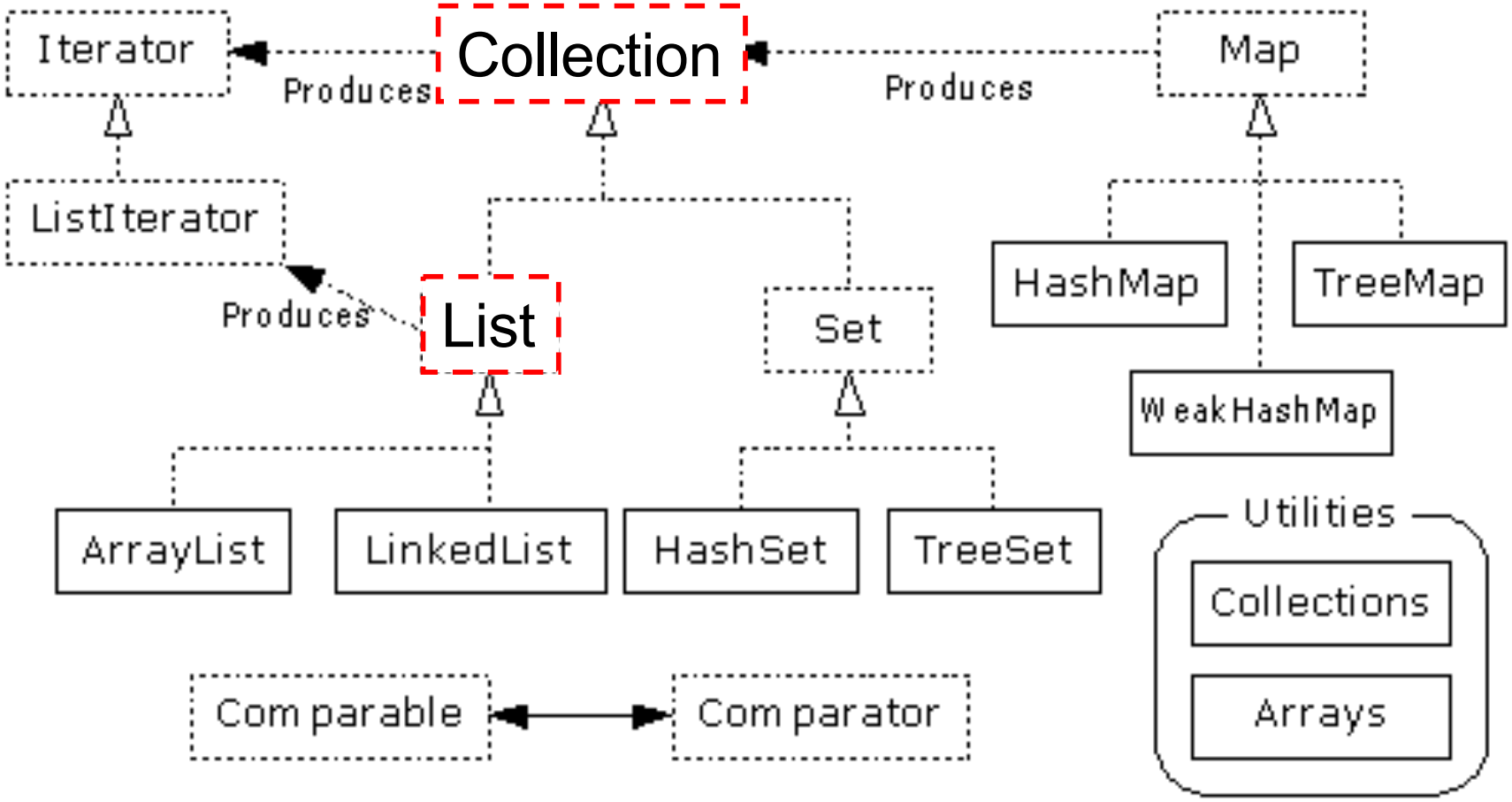


Advancing an iterator

Example - SimpleCollection

```
public class SimpleCollection  {
      public static void main(String[] args) {
         Collection c;
         c = new ArrayList();
         System.out.println(c.getClass().getName());
         for (int i=1; i <= 10; i++) {
                  c.add(i + " * " + i + " = "+i*i);
         }
         Iterator iter = c.iterator();
         while (iter.hasNext())
                  System.out.println(iter.next());
      }
}
```

List Interface Context



List Interface

- The List interface adds the notion of *order* to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow *duplicate* elements
- Provides a ListIterator to step through the elements in the list.

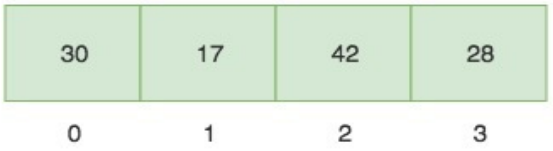
ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
  - **`void add(Object o)`** - before current position
  - **`boolean hasPrevious()`**
  - **`Object previous()`**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

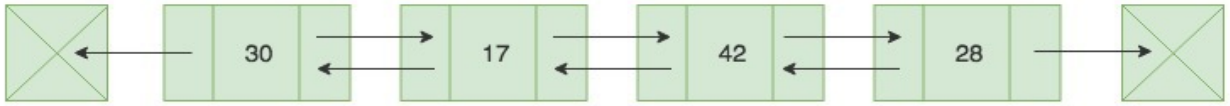
Two General-Purpose List Implementations

- ArrayList:
  - Simplest class that implements the List interface
  - The list stores the elements sequentially based on their index
  - This array has a capacity, dynamically expandable/shrinkable
- LinkedList:
  - uses a doubly-linked list to store its elements
  - A doubly-linked list consists of a collection of nodes, where each node contains three fields
    - The data at that node.
    - A pointer/reference to the next node in the list.
    - A pointer/reference to the previous node in the list.

Java ArrayList Representation



Java LinkedList Representation



ArrayList Overview

- Implements the List interface and uses an array as its *internal storage container*
  - It is a list, not an array
- The array that actual stores the elements of the list is hidden, not visible outside of the ArrayList class
  - all actions on ArrayList objects are via the methods
- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity
- Methods documentation:
  - https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Using ArrayList - Example Program

- This example shows:
  - How to create an ArrayList using the ArrayList() constructor.
  - Add new elements to an ArrayList using the add() method.
  - Get an element from an ArrayList using the get() method.

```
import java.util.ArrayList;
import java.util.List;

public class CreateArrayListExample {
   public static void main(String[] args) {
      // Creating an ArrayList of String
      List<String> animals = new ArrayList<>();

      // Adding new elements to the ArrayList
      animals.add("Lion");
      animals.add("Tiger");
      animals.add("Cat");
      animals.add("Dog");

      System.out.println(animals);

      // Adding an element at a particular index
      animals.add(2, "Elephant");
      System.out.println(animals);

      // Access an element at a particular index
      String firstAnimal = animals.get(0);
      System.out.println("The first animal: " + firstAnimal;
   }
}
```

LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
  - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
  - Start from beginning or end and traverse each node while counting
- Methods documentation:
  - https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Using LinkedList - Example Program

- The example shows
  - How to create a LinkedList using the LinkedList() constructor
  - How to add new elements to it using add(), addFirst() and addLast() methods.
  - How to get an elements using the get() method.

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
public class LinkedListDemo {
public static void main(String[] args) {
// Creating a LinkedList
LinkedList<String> friends = new LinkedList<>();

// Adding new elements to the end of the LinkedList using add() method.
friends.add("Rajeev");
friends.add("John");
friends.add("David");
friends.add("Chris");
System.out.println("Initial LinkedList : " + friends);

// Adding an element at the specified position in the LinkedList
friends.add(3, "Lisa");
 System.out.println("After add(3, \"Lisa\") : " + friends);

// Adding an element at the beginning of the LinkedList
friends.addFirst("Steve");
System.out.println("After addFirst(\"Steve\") : " + friends);

// Adding an element at the end of the LinkedList (This method is equivalent to the add() method)
friends.addLast("Jennifer");
System.out.println("After addLast(\"Jennifer\") : " + friends);

// Retrieving elements
String firstFriend = friends.get(0);
String firstFriend2 =friends.getFrist();
String lastFriend = friends.getLast();
System.out.println(" First friend : " + firstFriend);
System.out.println(" First friend : " + firstFriend2);
System.out.println(" Last friend : " + lastFriend);
```

Generic Types in Java

- The <E> notation is a placeholder for the element type used in the array. Class definitions that include a type parameter are called generic types.
  - Avoid significant amounts of type casting

- When you declare or create an ArrayList, it is a good idea to specify the element type in angle brackets.
  - Example: To declare and initialize an ArrayList called **names** that contains elements of type **String**, you would write:
    - **ArrayList<String> names = new ArrayList<String>();**

- Advantage
  - Java now knows what type of value the ArrayList contains.
  - When you call set, Java can ensure that the value matches the element type.
  - When you call get, Java knows what type of value to expect, eliminating the need for a type cast.

Autoboxing and Unboxing

- Generic types benefit substantially from the technique of autoboxing and unboxing
- As of Java Standard Edition 5.0, Java automatically converts values back and forth between a primitive type and the corresponding wrapper class.
  - This feature makes it possible to store primitive values in an ArrayList, even though the elements of any ArrayList must be a Java class.
  - Example:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(42);
int answer = list.get(0);
```

LinkedList vs. ArrayList

- LinkedList:
  - Linked lists may grow and shrink
  - Linear time access: O(n)
  - Linear time insertion and removal (except if previous element supplied, then constant): O(n)
- ArrayList:
  - ArrayList supports dynamic arrays that can grow as needed
    - Even though Arrays have a fixed size
  - Constant time access to elements: O(1)
  - Insertion at beginning or in the middle is linear: O(n)

algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
  - Sorting
  - Shuffling
  - Routine Data Manipulation
    - Reverse
    - Fill copy
    - etc.
  - Searching
    - Binary Search
  - Composition
    - Frequency
    - Disjoint
  - Finding extreme values
    - Min
    - Max

Terminologies

- A **collection** - an object that groups and represents multiple elements (typical other types of objects) as a single unit
- **JAVA Collection Framework** - a unified architecture for representing and manipulating collections, containing “Interfaces, Implementations, Algorithms”
- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.
- **Legacy Implementations** - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.
- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.