Collections

Part 1

Objectives

- Topics
  - Collection and Map interfaces and their concrete class implementations
  - Iterators and useful methods
- Goals: after this lecture, you will be able to
  - choose a Collection or Set implementation that matches a program's use cases
  - program applications using Collection and Set classes
  - create and use iterators
  - understand and use other built-in Collection and Set methods

Review

What does an Event Handler do?

a) Creates events like button presses

b) Responds to events like button presses

c) Enables a JavaFX program to create output

d) Lays out widgets like Buttons on the screen

Review

What is an anonymous inner class?

a) A class inside another class

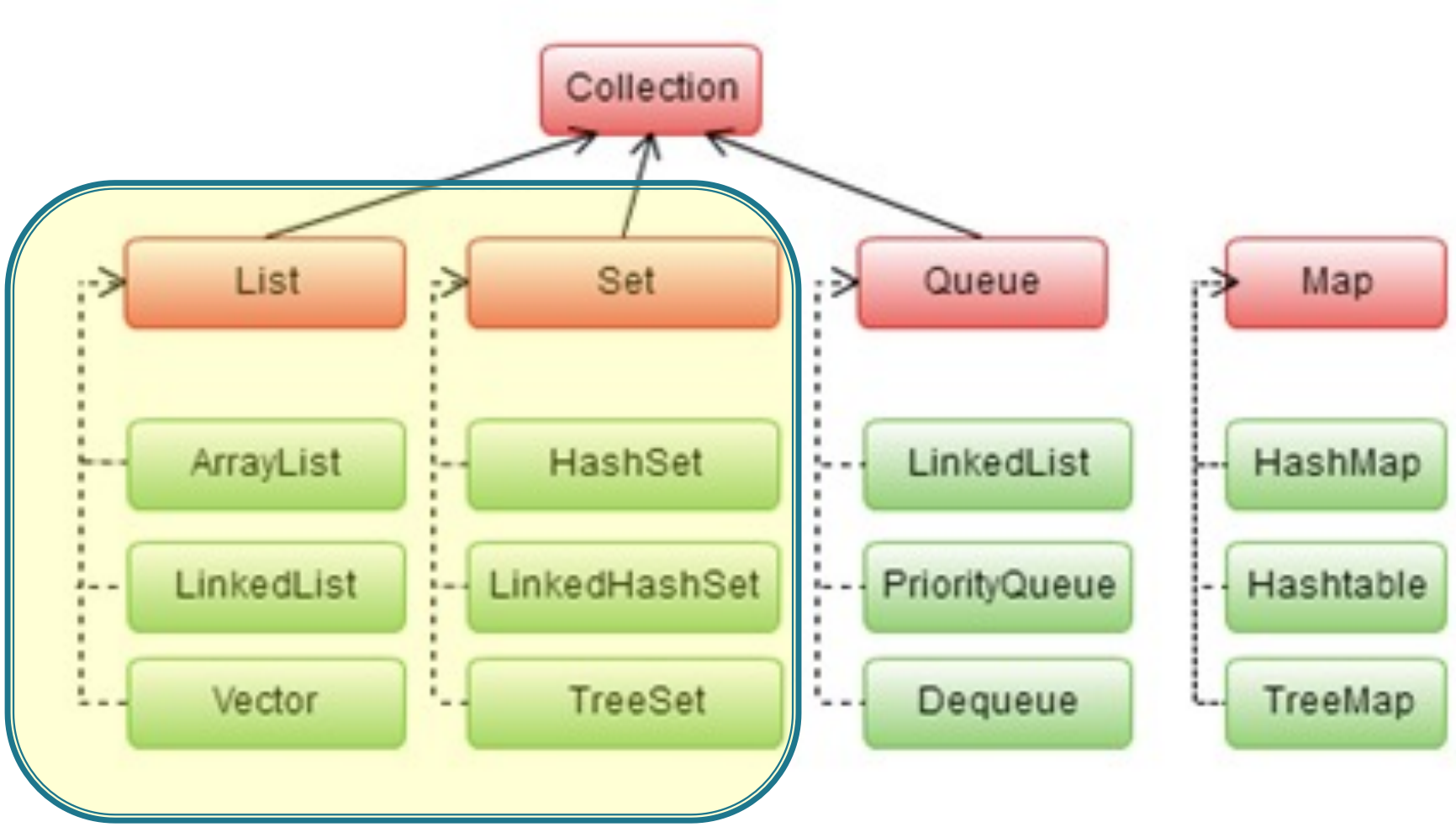
b) A class with no permanent name

c) A class new'd up on demand

d) All of the above

Collection and Map

- ***Collection*** is an interface with three basic children:
  - ***List*** for linear collections: implemented by ArrayList, LinkedList
  - ***Set*** for no-duplicate collections: implemented by HashSet, TreeSet, LinkedHashSet
  - ***Queue*** for first-in, first-out collections: implemented by LinkedList, PriorityQueue, Deque
- All are collections of objects, ***not*** primitive types
- ***Map*** is another interface, used for hash map types



https://fresh2refresh.com/java-tutorial/java-collections-framework/

List

- The **`List<>`** interface keeps elements in some order
  - Interface, not class, so new up as one of the following
- **`ArrayList<>`** generalizes an array, allowing it to grow and shrink but keeping the index-based lookup. Insertion or deletion in the middle of the ArrayList is slow
- **`LinkedList<>`** is also expandable, no index lookup, but middle insertion/deletion is fast
- **`Vector<>`** is an alternative to ArrayList used with threading

ArrayList

- Style is to name the variable by interface, new it by class:

```
List<String> list = new ArrayList<>();
```

- but you can say:

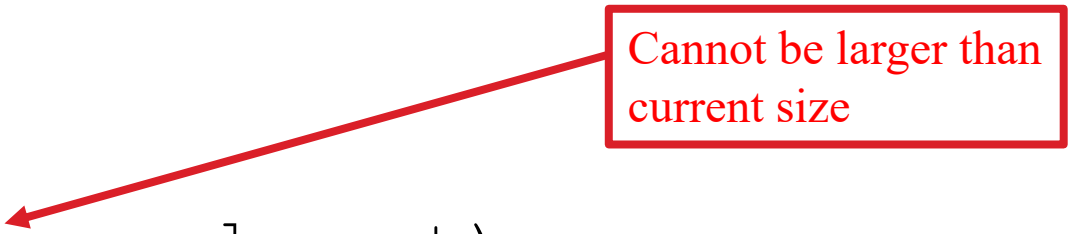
```
ArrayList<String> list = new ArrayList<>();
```

- 10 elements by default, unless you ask for more:

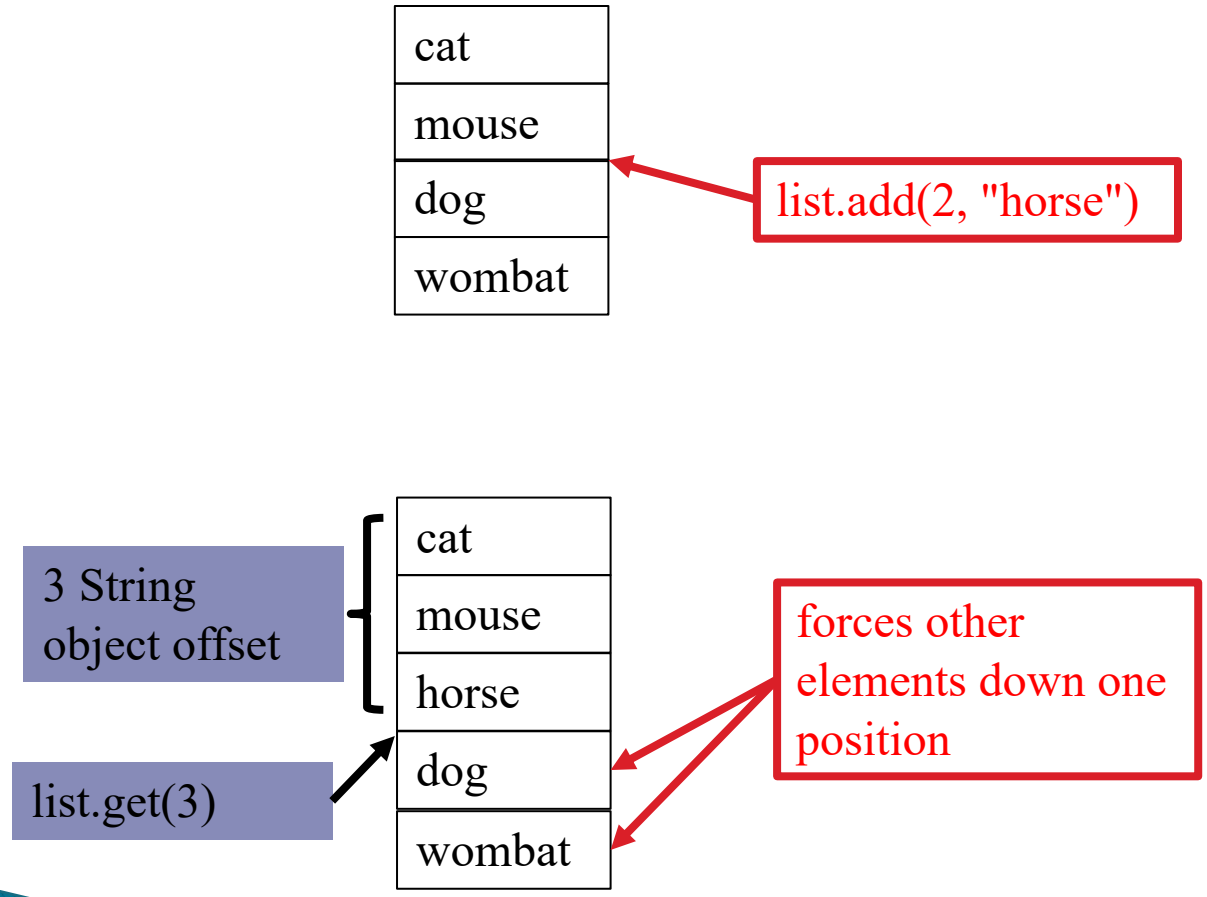
```
List<String> list = new ArrayList<>(50);
```

ArrayList

- Some methods:
  - `add(element), add(index, element), addAll(ArrayList)`
  - `contains(element), indexOf(element)`
  - `get(index), remove(index), remove(element)`
  - `size(), isEmpty()`
  - `toArray(), subList(from, to)`
  - and `foreach` loops



ArrayList is expandable, at a cost

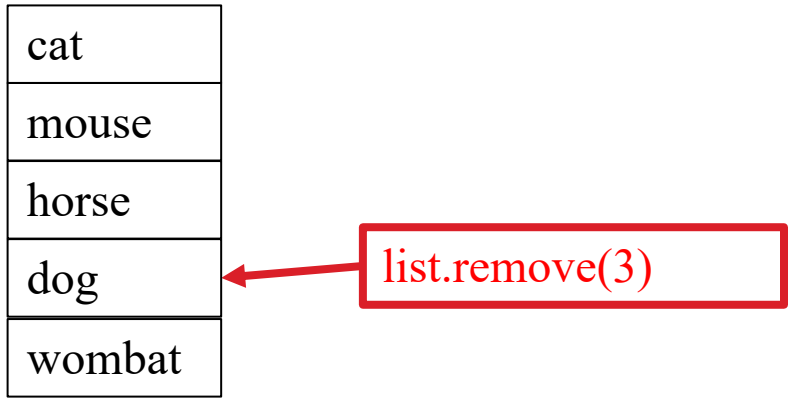


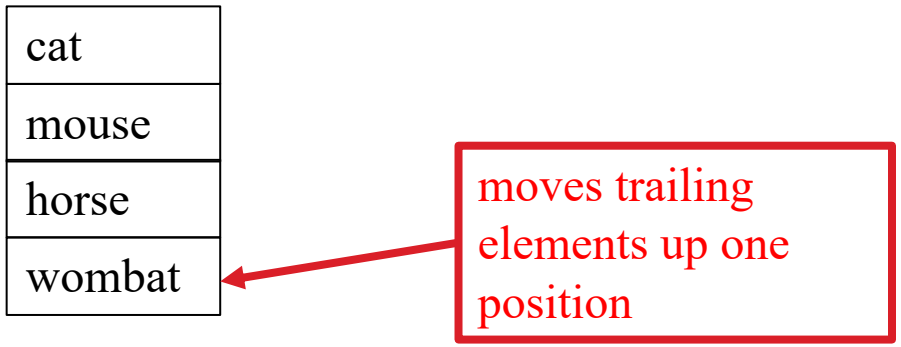
- Insertion is O(n) because on average, move half the elements
- In addition, you may trip the size boundary, requiring memory allocation and copying

ArrayList, cont.

- Deletion is similar: move elements up so there's no "hole" in the array
  - Why not just leave it blank, or mark it as deleted? There'd be a lot of index adjustments needed
  - And besides, arrays and ArrayLists are supposed to use contiguous memory for speed of lookup

ArrayList, cont.

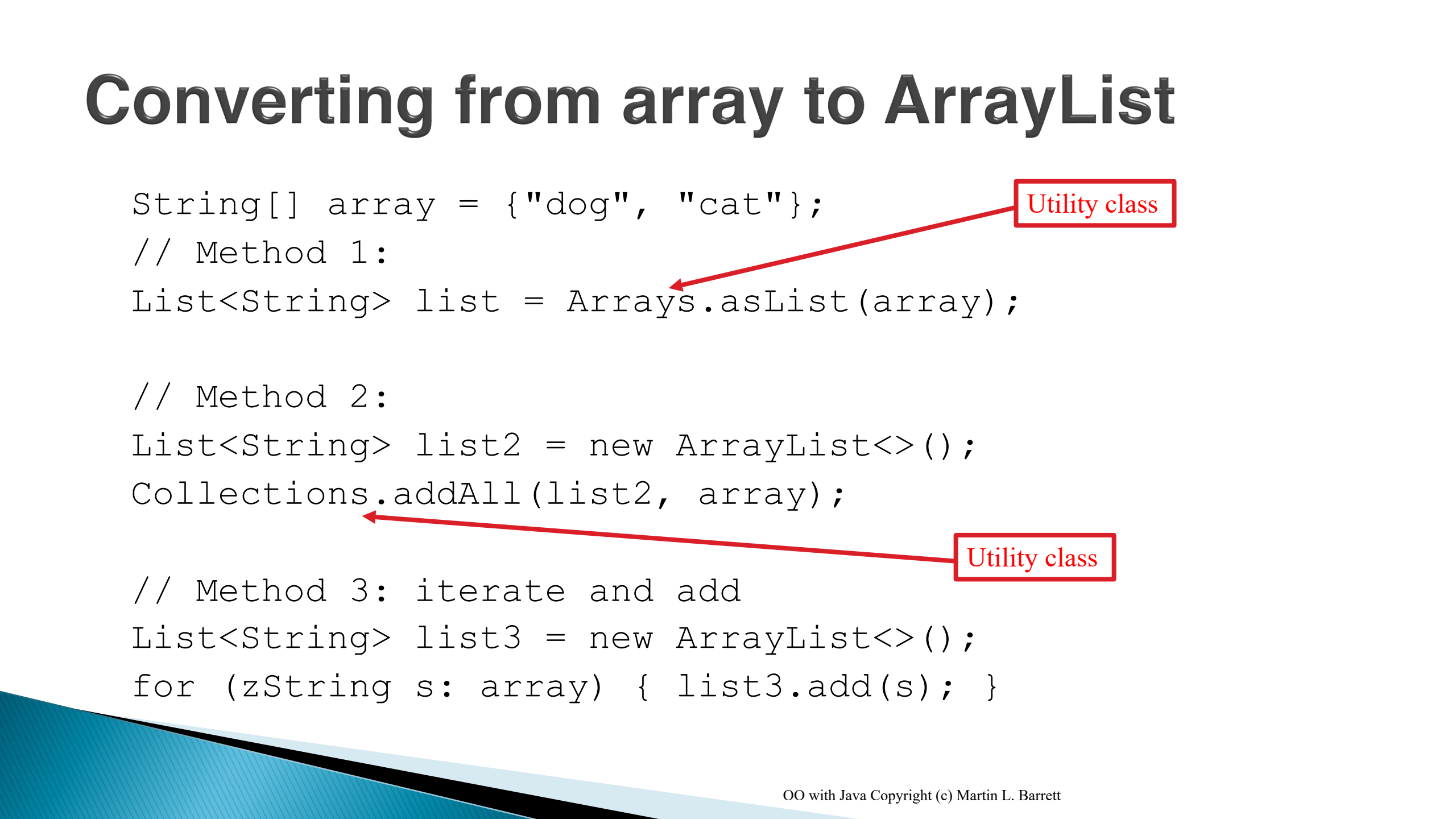




- Deletion also incurs O(n) time because of moving elements: on average, you'll have to move up about half the elements
- Can also trip the size boundary, causing more copying

Converting from array to ArrayList

```
String[] array = {"dog", "cat"};
// Method 1:
List<String> list = Arrays.asList(array);

// Method 2:
List<String> list2 = new ArrayList<>();
Collections.addAll(list2, array);

// Method 3: iterate and add
List<String> list3 = new ArrayList<>();
for (zString s: array) { list3.add(s); }
```



Converting from ArrayList to array

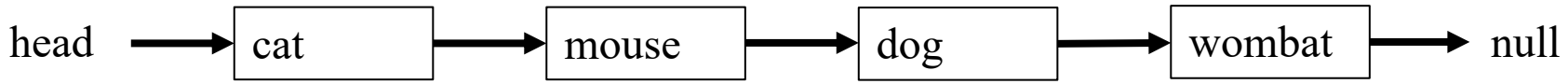
```
List<String> list4 = new ArrayList<>();
list4.add("dog"); list4.add("cat");

String[] array2 = list4.toArray( String[0] );
```

LinkedList

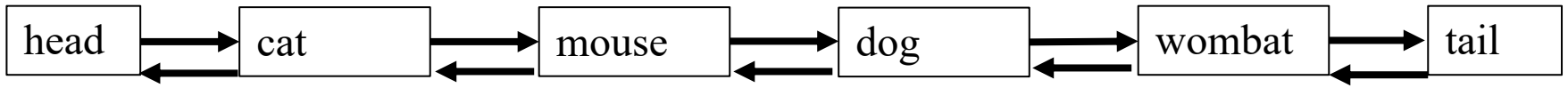
- Same declaration style, same operations

```
List<String> list = new LinkedList<>();
```

- In general, a linked list can be ***singly linked***:



- or ***doubly linked***:

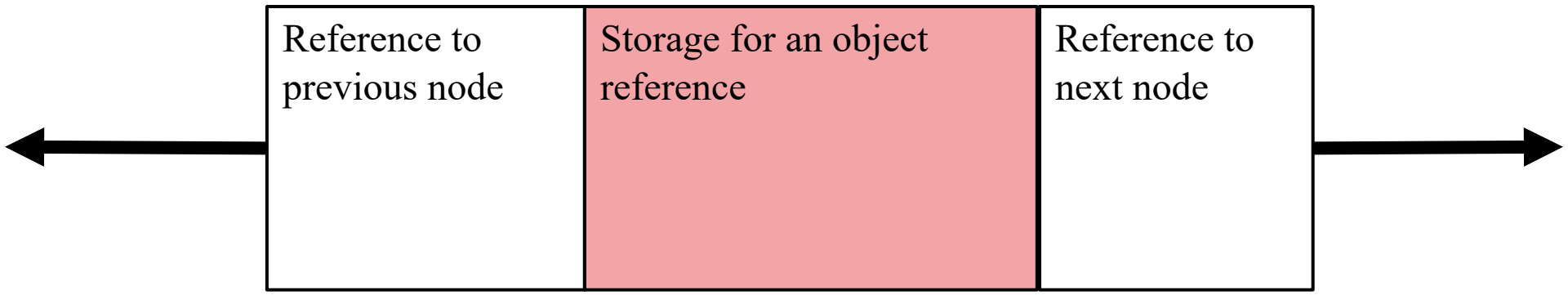


LinkedList, cont.

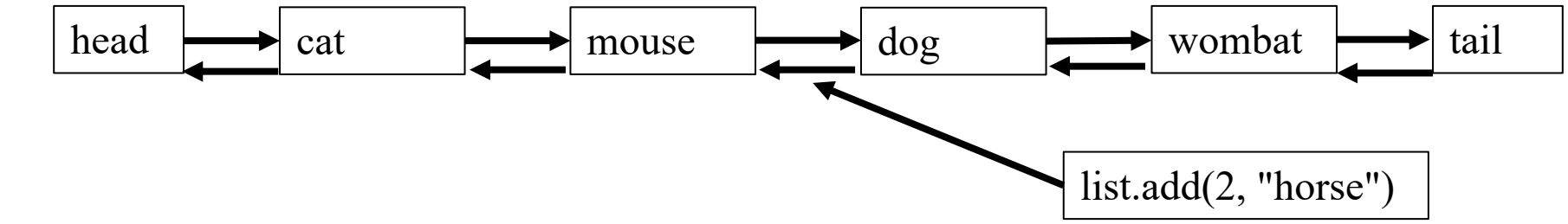
- Each implementation has a next pointer; doubly linked also has previous pointer
- Java uses doubly linked
  - That makes is slightly slower for some operations
  - but it has no coding implications
- All List< > methods apply here, too
  - so add( ), remove( ), etc.

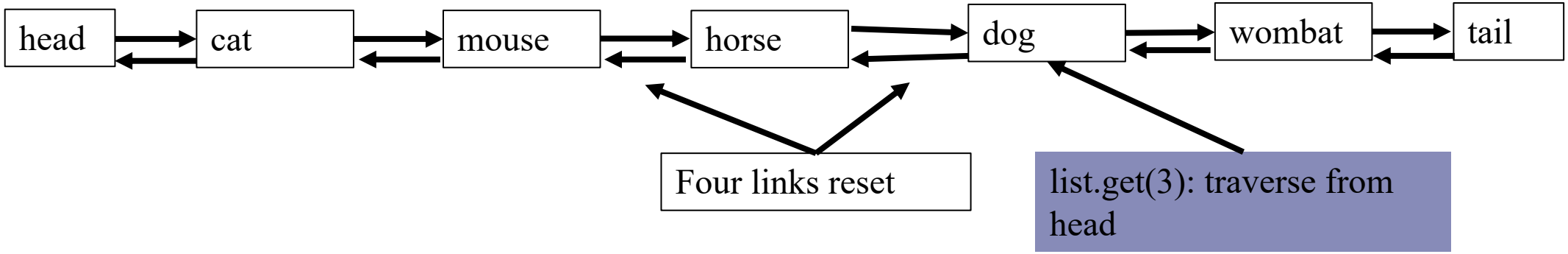
LinkedList node

- To use a LinkedList, you don't need to know how it's implemented – this is just for background knowledge



LinkdedList Insertion



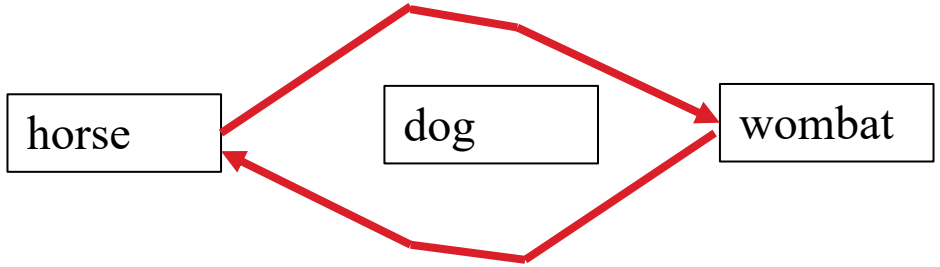


- Insertion and deletion are very efficient (reference resets), but get( ), contains(), indexOf() are not – require traversals

LinkedList Deletion

- Deletion is similar: reset the references to point around the deleted node, its memory will eventually be garbage collected


```
remove("dog")
```



ArrayList or LinkedList?

- Both are linear structures
  - Search is O(n)
- But other operations vary: for example, insert at front:
  - LinkedList is O(1), but ArrayList is O(n)
- ArrayList is better when most/all of the data is inserted at once, with few insertions/deletions later
- LinkedList is better when there will be many insertions and/or deletions later
- ArrayList is faster for sorting followed by searching

Set

- `Set< >` is an unordered collection of objects, with no duplicates
- There are fewer use cases for Set.
  - Its main draw is preventing duplicate entries
- Three implementaions:
  - `TreeSet< >`
  - `LinkedHashSet< >`
  - `HashSet< >`
- First two maintain sorted ordering, but HashSet does not
  - So that "unordered collection" thing …

Set, cont.

```
Set<String> set = new TreeSet<>();
set.add("dog"); set.add("cat"); set.add("dog"); set.add("horse");
// Traversing this prints: cat dog horse

Set<String> set2 = new HashSet<>();
set2.add("dog"); set2.add("cat"); set2.add("dog"); set2.add("horse");
// Traversing this prints: horse cat dog
```

Iterator<>

- Set does not have a get() method, but you can iterate over them with:
  - foreach loop
  - ***Iterator***: interface that abstracts "walking through" a Collection (not just a Set). Defines next(), hasNext(), remove()

```
Set<String> set = new TreeSet<>();
set.add("dog"); set.add("cat"); set.add("dog"); set.add("horse");

// Set.iterator() returns the Iterator object
Iterator<String> iter = set.iterator();
while ( iter.hasNext() ) {
      String s = iter.next();    // Move to the next one
      System.out.println( s );   // Displays cat, dog, horse
}
```

Set, cont.

- The union and intersection operations can be simulated with Set
  - Union: addAll()
  - Intersection: retainAll()

```
Set<String> result = new TreeSet<>(set);     // Starts with contents of set
result.addAll(set2);                         // Union: add, but no duplicates

Set<String> result2 = new TreeSet<>(set);    // Again starts with contents of set
result2.retainAll(set2);                     // Intersection: keep only what's in both
```

Example: remove duplicates from a List

```
List<String> list = new ArrayList<>();
list.add("cat"); list.add("mouse"); list.add("dog"); list.add("wombat");
list.add("dog"); list.add("dog");
// Contains: cat mouse dog wombat dog dog

Set<String> myset = new TreeSet<>(list);// Convert to Set
// Contains: cat dog horse mouse wombat

List<String> newlist = new ArrayList<>(myset); // Convert back to List
// Contains: cat dog horse mouse wombat

// or all at once:
List<String> newlist = new ArrayList<>( new TreeSet<>(list) );
```

Collection Utility Methods

- `Collections.sort(list), Collections.sort(list, comparable)`
- `Collections.binarySearch(key)` – only after sorting!
- `Collections.max(list), Collections.max(list, comparable)`
- `Collections.min(list), Collections.min(list, comparable)`
- `Collections.reverse(list)`
- `Collections.frequency(list, thing)`

Collection Utility Methods

```
List<String> list = new ArrayList<>();
list.add("aardvark"); list.add("gopher");
list.add("zebra");
Collections.reverse( list );
Iterator<String> iterator = list.iterator();
while ( iterator.hasNext() ) {
    System.out.println( iterator.next() );
}
// Prints: zebra, gopher, aardvark
```