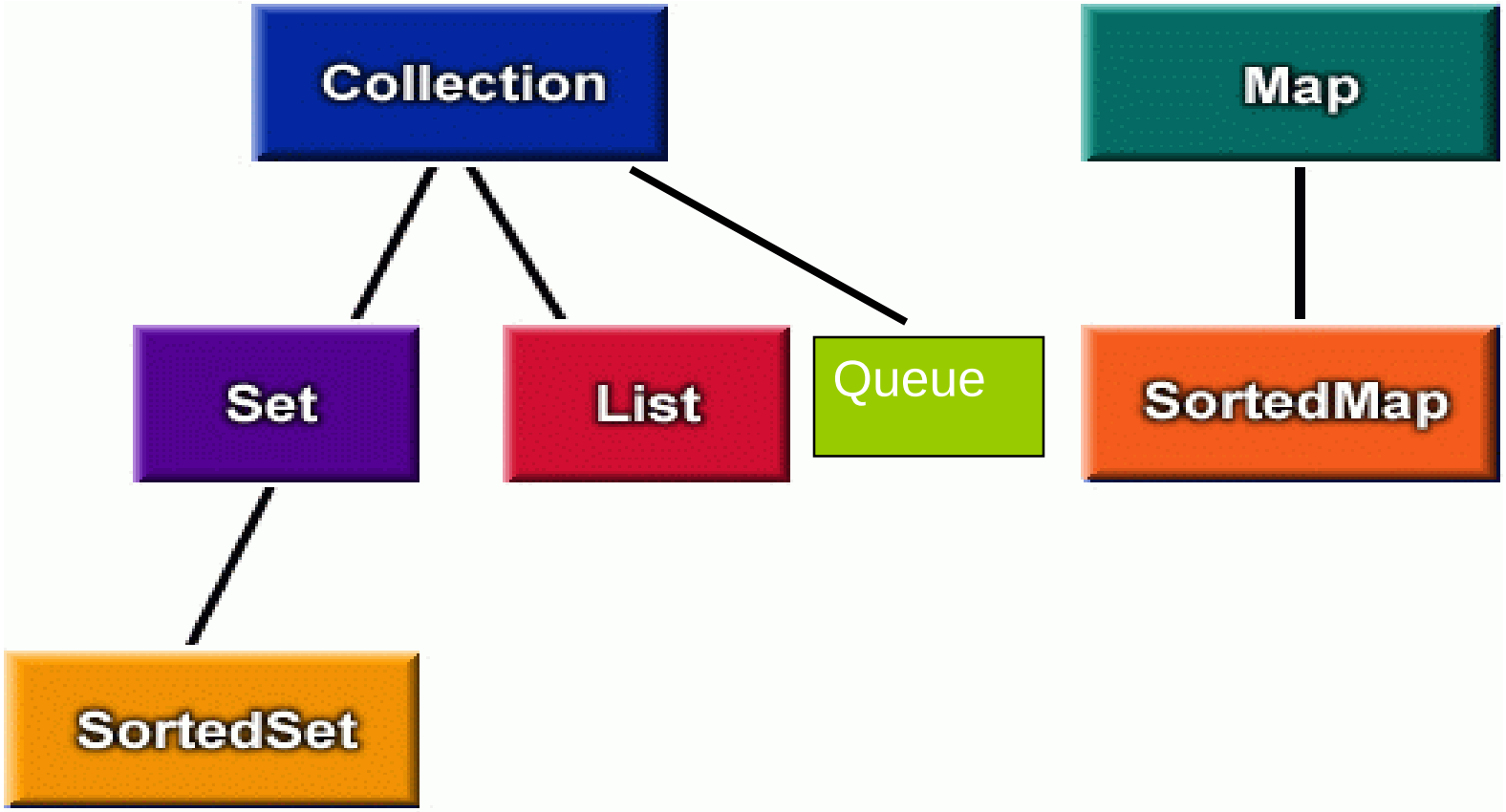
Java Collection Framework

Collection Framework

- A *collection framework* is a unified architecture for representing and manipulating collections. It has:
  - **Interfaces:** abstract data types representing collections
  - **Implementations:** concrete implementations of the collection interfaces
  - **Algorithms:** methods that perform useful computations, such as searching and sorting
    - These algorithms are said to be *polymorphic*: the same method can be used on different implementations

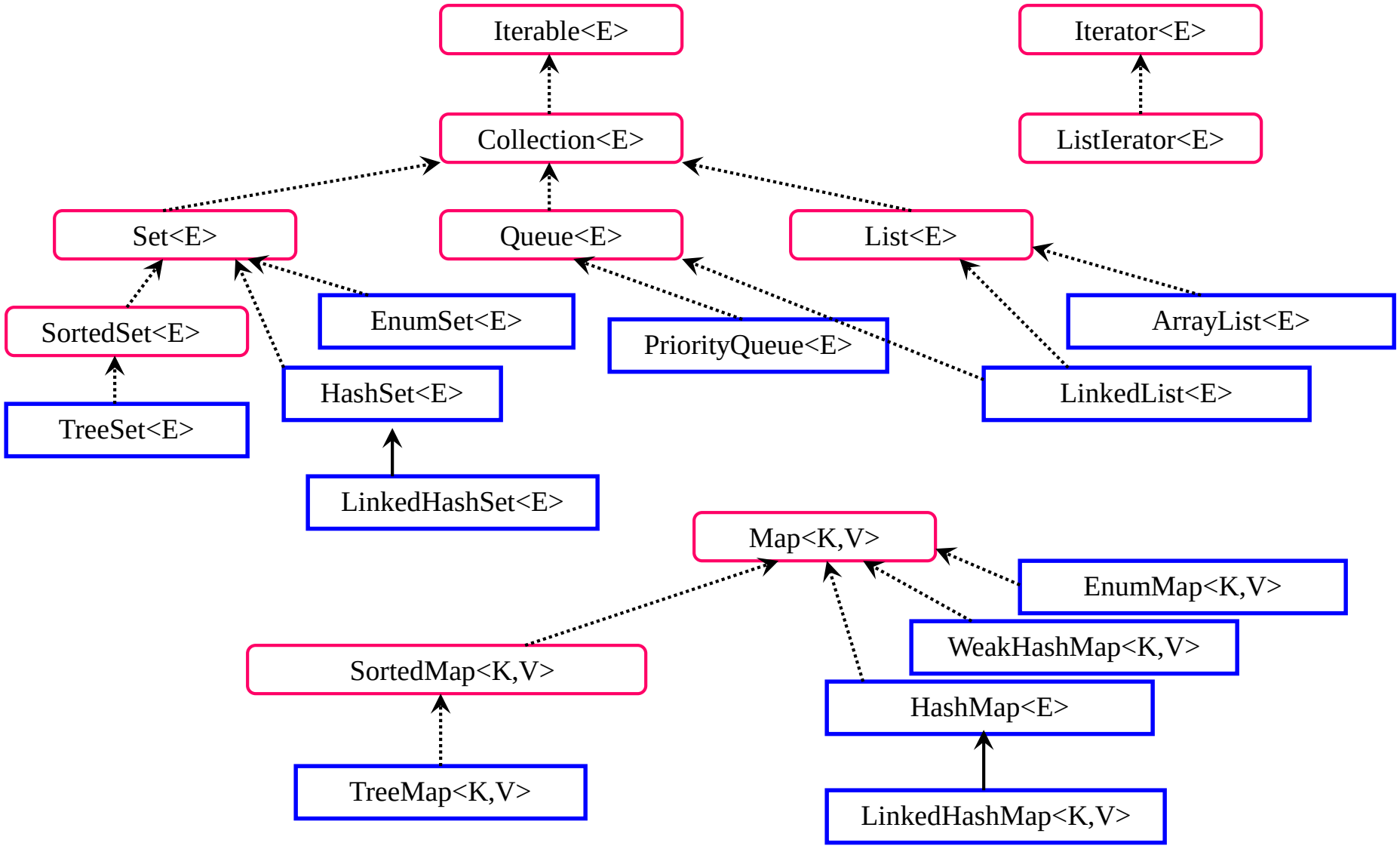
Collection interfaces



Collection Interface continued

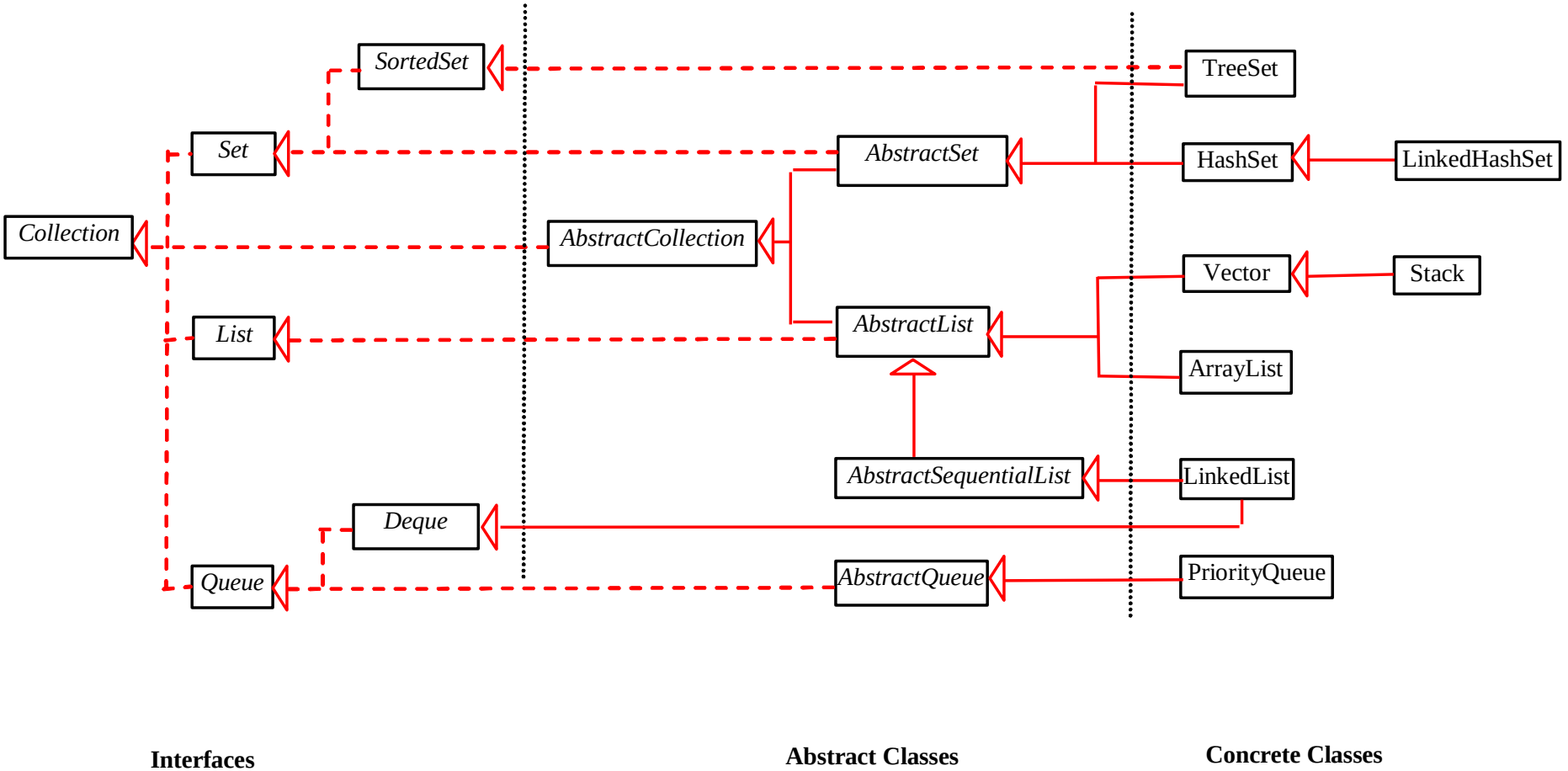
- **Set →**
  - The familiar set abstraction.
  - No duplicates; May or may not be ordered.
- **List →**
  - Ordered collection, also known as a sequence.
  - Duplicates permitted; Allows positional access
- **Map →**
  - A mapping from keys to values.
  - Each key can map to at most one value (function).
  - The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.
- **Queue →**
  - Ordered collection. FIFO (First In First Out)

Type Trees for Collections

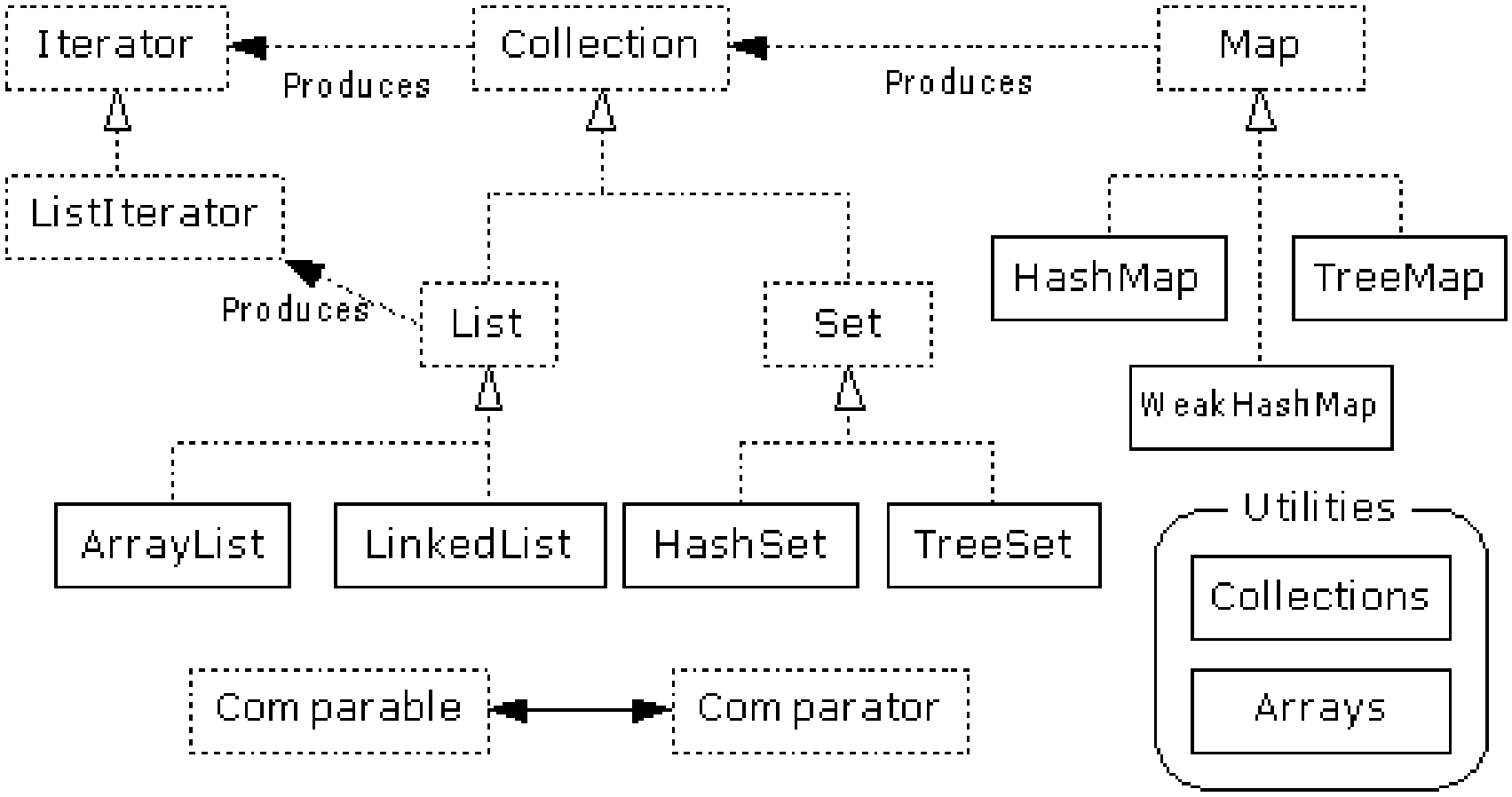


Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.



Collections Framework Diagram



Collection Interface

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

Interface `Collection`

- `add(o)` Add a new element
- `addAll(c)` Add a collection
- `clear()` Remove all elements
- `contains(o)` Membership checking.
- `containsAll(c)` Inclusion checking
- `isEmpty()` Whether it is empty
- `iterator()` Return an iterator
- `remove(o)` Remove an element
- `removeAll(c)` Remove a collection
- `retainAll(c)` Keep the elements
- `size()` The number of elements

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

Iterator Position

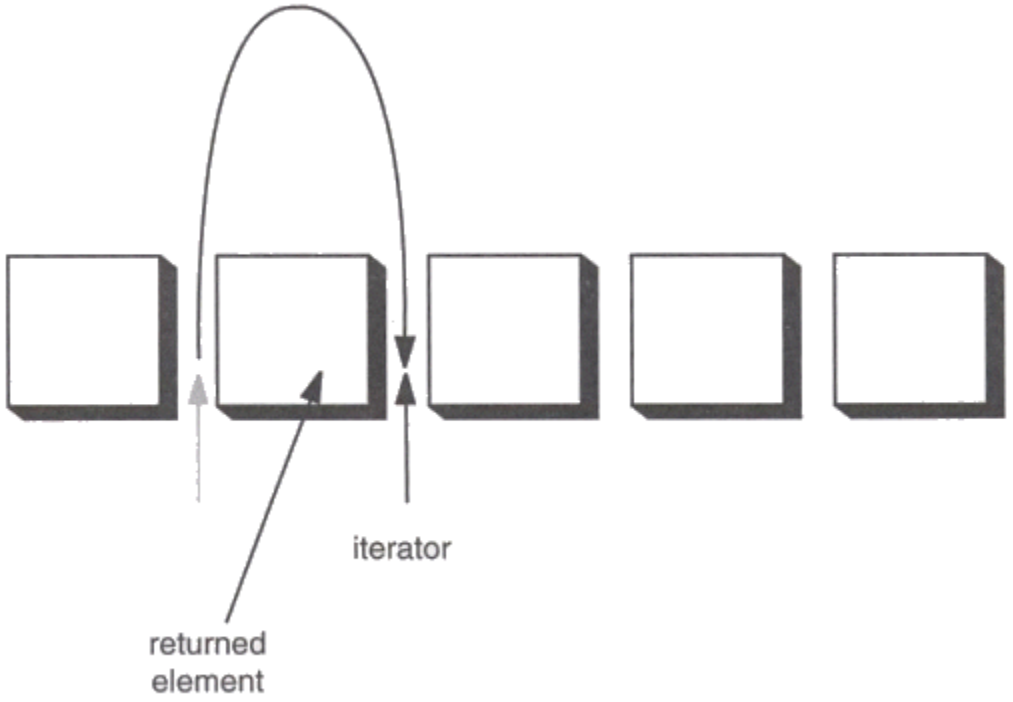
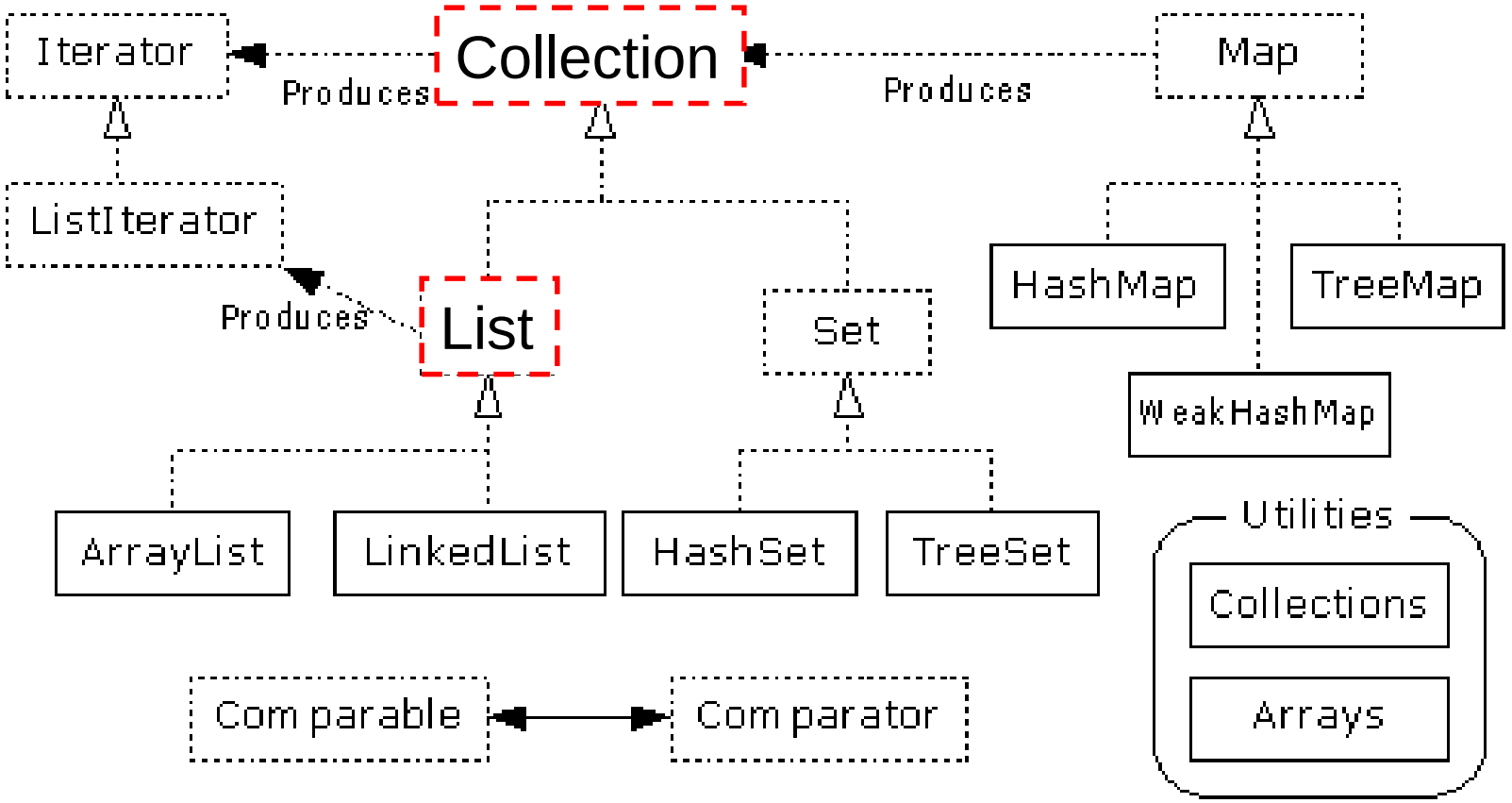


Figure 2–3: Advancing an iterator

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

Iterator Position - `next()`, `previous()`

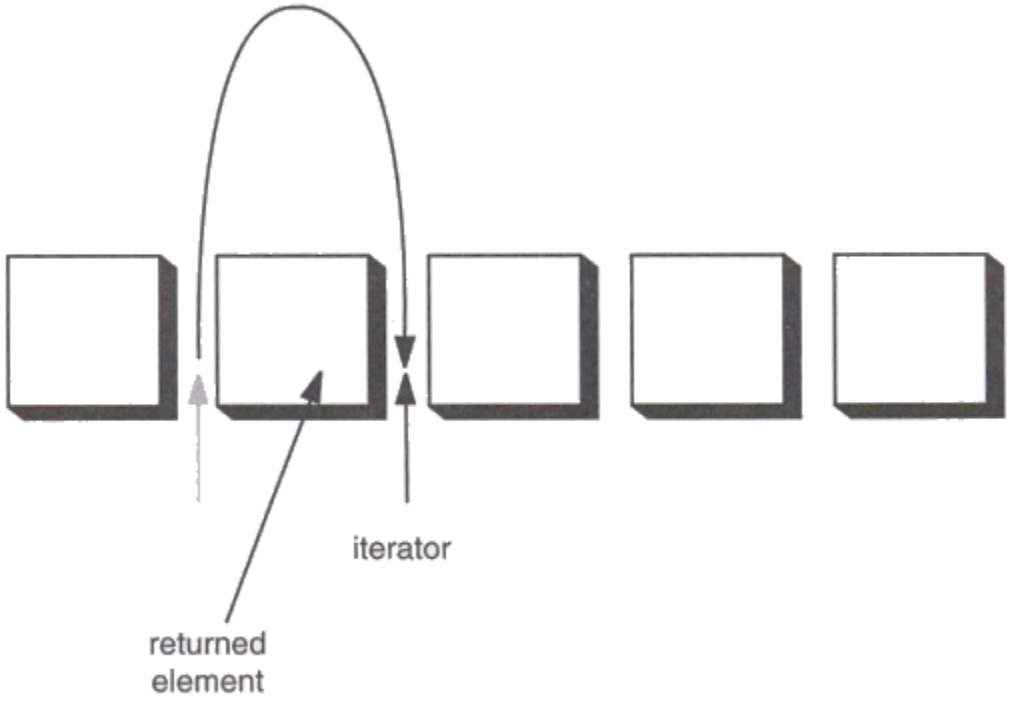
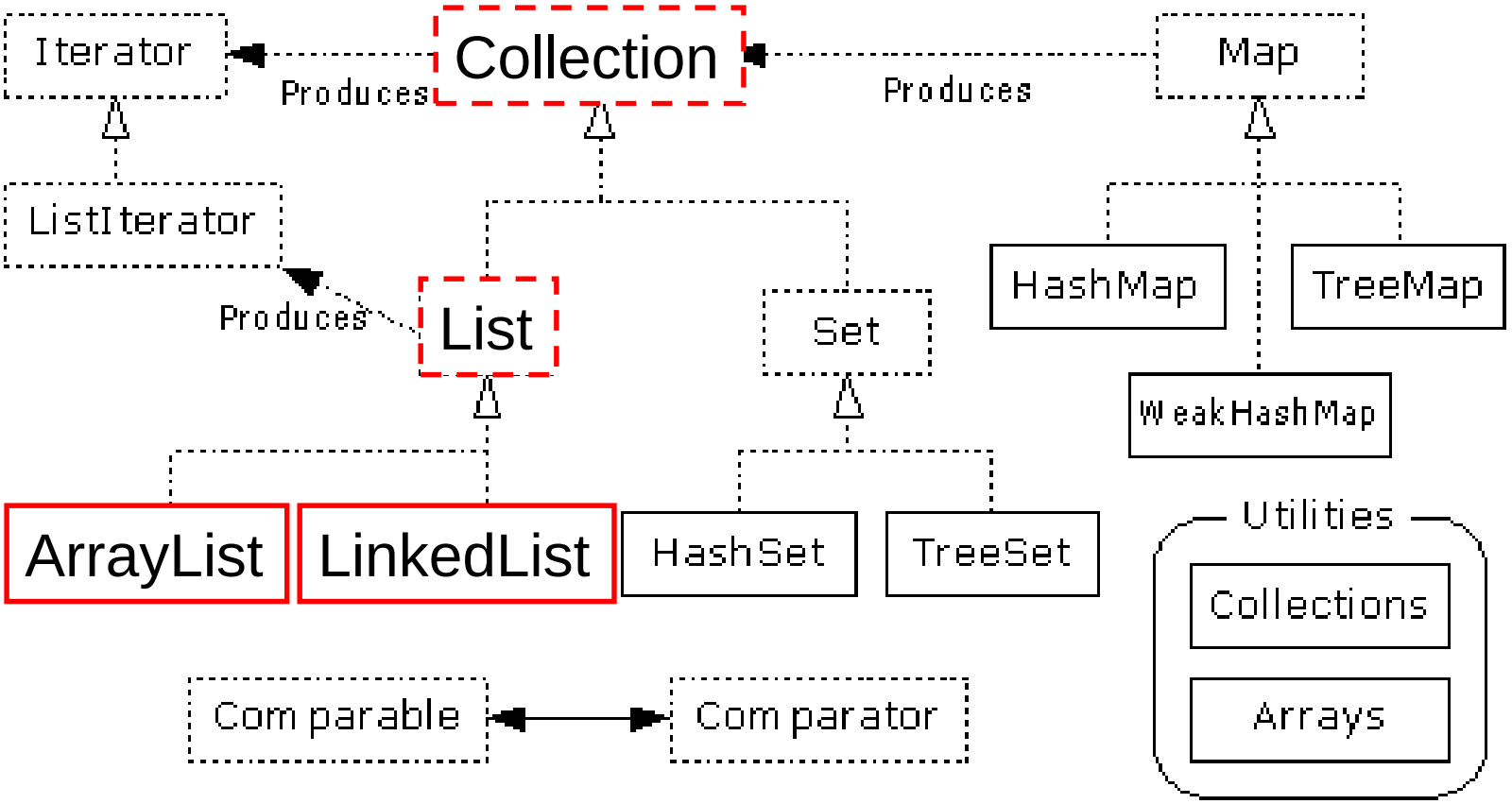


Figure 2–3: Advancing an iterator

ArrayList and LinkedList Context



List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access

ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException(
                "Illegal Capacity: "+initialCapacity);
    this.elementData = new Object[initialCapacity];
}
```

ArrayList methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - **Object get(int index)**
  - **Object set(int index, Object element)**
- Indexed add and remove are provided, but can be costly if used frequently
  - **void add(int index, Object element)**
  - **Object remove(int index)**
- May want to resize in one shot if adding many elements
  - **void ensureCapacity(int minCapacity)**

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

LinkedList methods

- The list is sequential, so access it that way
  - **ListIterator listIterator()**
- ListIterator knows about position
  - use **add()** from ListIterator to add at a position
  - use **remove()** from ListIterator to remove at a position
- LinkedList knows a few things too
  - **void addFirst(Object o), void addLast(Object o)**
  - **Object getFirst(), Object getLast()**
  - **Object removeFirst(), Object removeLast()**