Sequential Data Structures (v5)

Juan Manuel Gimeno Illa

2025/2026

Bibliography

- Maurice Naftalin & Philip Wadler with Stuart Marks, Java Generics and Collections (2nd edition), O'Reilly Media Inc. (2025)
  - Chapters 14, 13 (Deque)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein, Introduction to Algorithms – Fourth Edition, Massachusetts Institute of Technology (2022)
  - Chapter 10
- Robert Sedgewick & Kevin Wayne, Algorithms - Fourth Edition, Pearson Education Inc. (2011)
  - Chapter 1
- Josep Maria Ribó, Apropament a les estructures de dades des del programari lliure. Edicions de la Universitat de Lleida. (2018).
  - Chapter 2
- Benjamin J. Evans, Jason Clark & David Flanagan, Java in a Nutshell (8th edition), O'Reilly Media Inc. (2023)
  - Chapter 4 (nested classes)
- The Java Tutorials (nested classes)

Data Structures

- A **data structure** is a way to **organize the data** involved in a program
  - It must provide the **operations** the program needs
  - With a reasonable **cost** (constant, logarithmic, linear, linearithmic)
- For us, the study of data structures will have different legs
  - The **data structure itself** (theory, independent of the language)
  - Its **implementation in Java** (so, we'll need some java specific elements)
  - And its **real implementation** in the **Java Collections Framework (JCF)**
    - So, we'll study real and very well engineered code
- And we will need to understand **all these three things** !!

Why the implementations in the JCF?

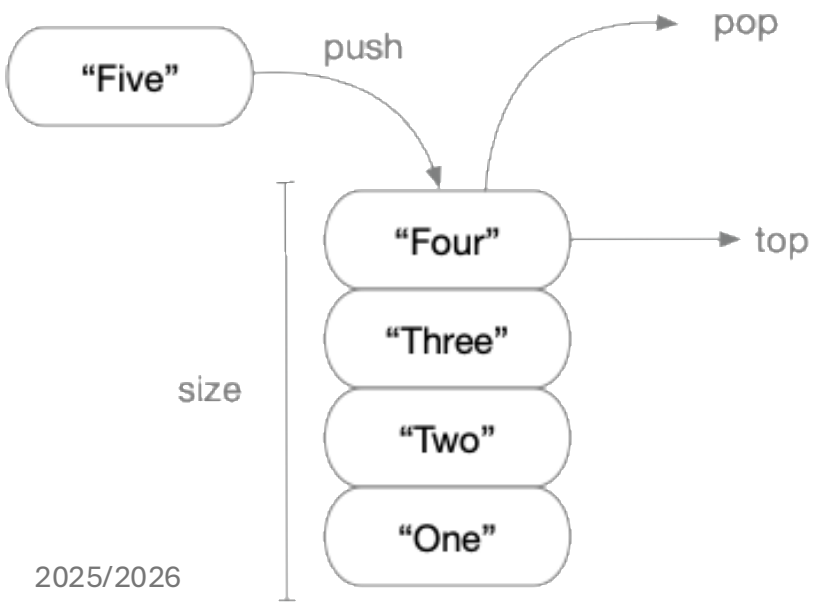
- They are the **most used** in Java, because it belongs to the Java standard library
  - So, in our daily Java programming, we'll problably use the JCF classes & interfaces
- So, knowing well the JCF will **improve** our Java **programming skills**
- If you want to consider other libraries for data structures in Java, you can read the TFG: Comparativa entre diferentes librerías de Estructuras de Datos en Java, by José Ramón Ariza Pérez

Sequential Data Structures

- A **sequential** data structure provides **insertion/deletion/access** to the elements by **position**
  - **First** (usually 0), second, …, **last**
  - The difference among them are the **accessible positions** and the **available operations**
- **Stack**:
  - We can insert/delete/access only at **one end**
- **Queue**:
  - We can **insert** at **one end**, and **delete/access** at the **the other end**
- **Deque**:
  - We can **insert/delete/access** at **both ends**
- **List**:
  - We can **insert/delete/access** at **any position**

Stack

```
public interface MyStack<E> {
   void push(E element);
   void pop();
   E top();
   int size();
   default boolean isEmpty() {
      return size() == 0;
   }
}
```



- I've named the type MyStack<E> to not confound it with the predefined Stack<E>
  - the class Stack<E> in JCF is **deprecated**
  - **one should use a Deque<E> instead**
- **Push**:
  - adds the given element at the top of the stack
- **Pop**:
  - removes the element at the top
- **Top**:
  - Returns the element at the top
- **Size**:
  - Returns the number of elements in the stack
- **NOTE**: We leave their implementations as exercises because they are a reduced set of operations on lists.

Lists

- A **list** is a data structure in which **each element has a position in the list**, and insertions, deletions and accesses can be done at **any position**
- We will
  - define a very minimal interface on lists
  - present a simple implementation in Java using (extensible) arrays
  - describe the real implementations in the JCF, which use two different approaches (with different costs)
- And, in doing so, we also will
  - present iterators on lists
  - add nested (and inner) classes to our knowledge of Java

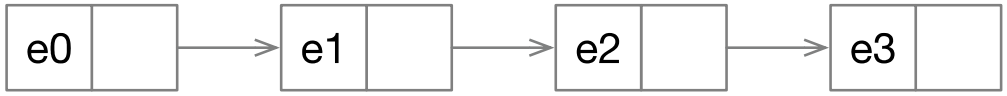
Lists

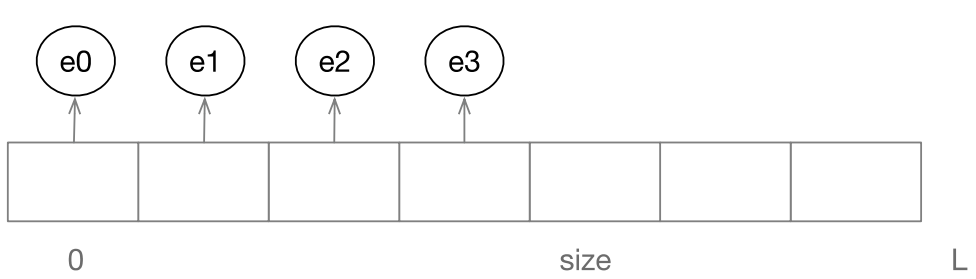
```
public interface MyList<E>
      extends Iterable<E> {

   void add(int index, E element);
   E set(int index, E element);
   E get(int index);
   void remove(int index);

   int size();

   Iterator<E> iterator();

   default boolean isEmpty() {
      return size() == 0;
   }
}
```

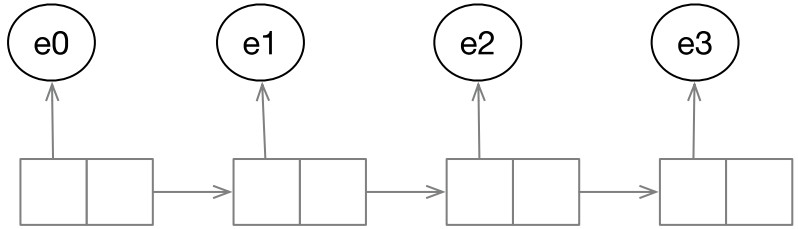
- We have defined MyList<E> as an Iterable<E>, to be able to use lists in foreach loops.
- Valid indexes:
  - add: $[0, size()]$
  - set/get/remove: $[0, size())$
  - If the index is not valid, the method throws IndexOutOfBoundsException
- set returns the old value at this index
- iterator creates an iterator that will access the elements from index 0 to $size() - 1$

List

- Also, the lists we'll implements have an **unbounded size**
- There are **two different ways** to implement this type:
  - Using an (extensible) **array** of **elements**
  - Using a **linked** structure of **nodes**

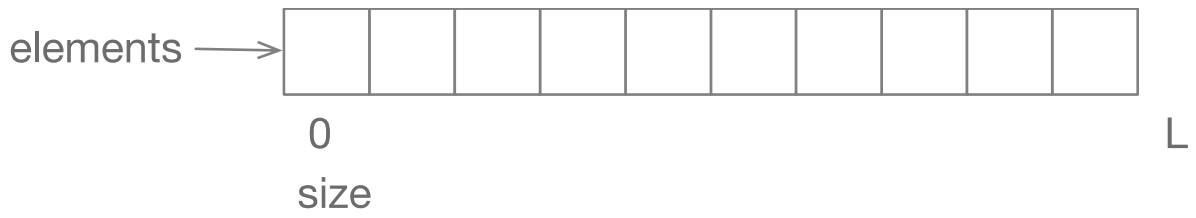






ArrayList

```
public class MyArrayList<E> implements MyList<E> {

  private static final int DEFAULT_SIZE = 10;

  private Object[] elements;
  private int size;
```



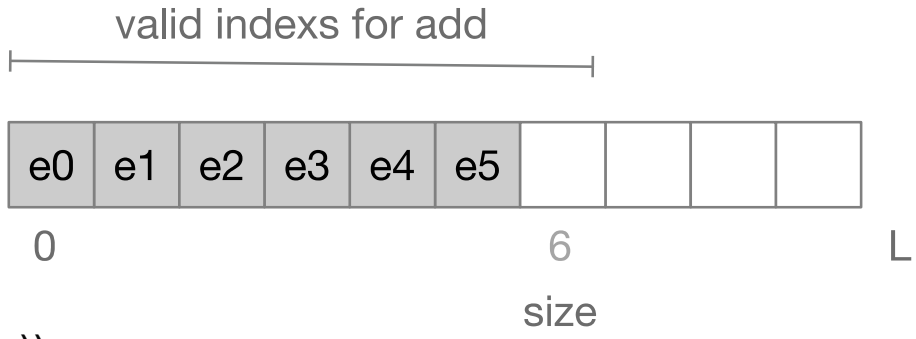

```
  public MyArrayList() {
    elements = new Object[DEFAULT_SIZE];
    size = 0;
  }
```

- Our first strategy for the implementation will be to use a resizable array to hold the elements of the list
- In this implementation the elements of the list are the elements of the prefix of the array $[0, size)$
- It's an array of Object as we cannot create an array of E's.

ArrayList

```
@Override
public void add(int index, E e) {
   checkIndexForAdd(index);
   ensureCapacity();
   moveSuffixOneRight(index);
   elements[index] = e;
   size += 1;
}
```

1. We check if the index is valid
2. We ensure that we have enough capacity in the array for the new element
3. We create a hole at index, moving the elements in the suffix [index, size) one position to the right
4. We add the element to the array at position index
5. We increment the size of the list

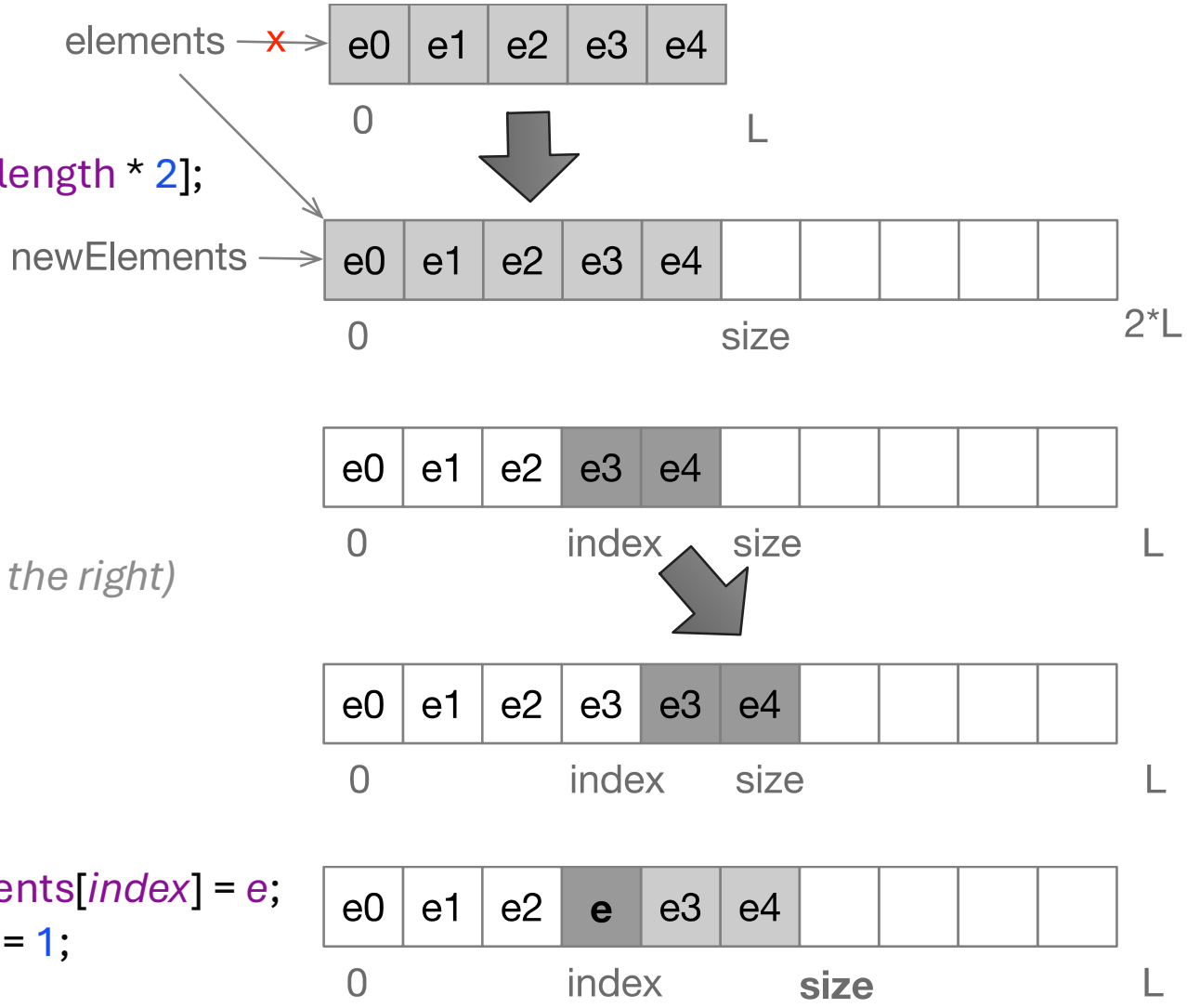



```
private void checkIndexForAdd(int index) {
   if (index < 0 || index > size)
      throw new IndexOutOfBoundsException(
            "adding at %d for a list of size %d".formatted(index, size));
}
```

ArrayList

```
private void ensureCapacity() {
   if (size < elements.length) return;
   Object[] newElements = new Object[elements.length * 2];
   for (int i = 0; i < size; i++) {
      newElements[i] = elements[i];
   }
   elements = newElements;
}

private void moveSuffixOneRight(int index) {
   // loop from right to left (the suffix moves one to the right)
   // index is the destination
   for (int i = size; i > index; i--) {
      elements[i] = elements[i - 1];
   }
}
```

```
elements[index] = e;
size += 1;
```

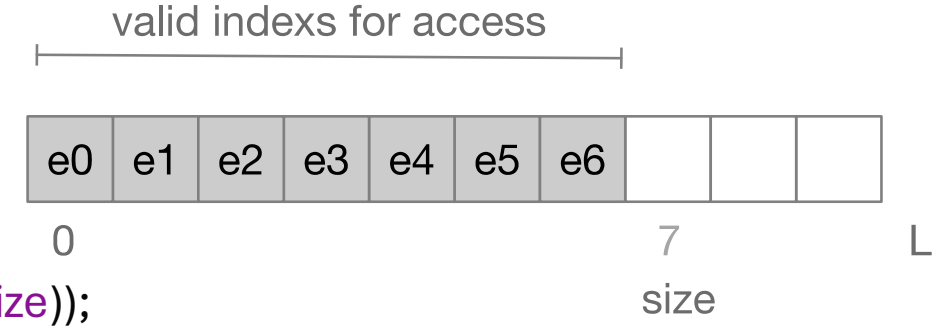


ArrayList

```
@Override
public E set(int index, E element) {
   checkIndexForAccess(index);
   @SuppressWarnings("unchecked")
   E oldElement = (E) elements[index];
   elements[index] = element;
   return oldElement;
}
```

1. We check the index is valid
2. We get hold of the old value at the index position
   - The array is of Objects, but we need to return an E, so we cast
   - But the compiler cannot check this cast is OK (we can because the only things we have added are E's)
   - So, we tell the compiler to not issue a warning
3. We substitute with the new value
4. We return the old one

```
private void checkIndexForAccess(int index) {
   if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException(
            "accessing at %d for a list of size %d".formatted(index, size));
}
```



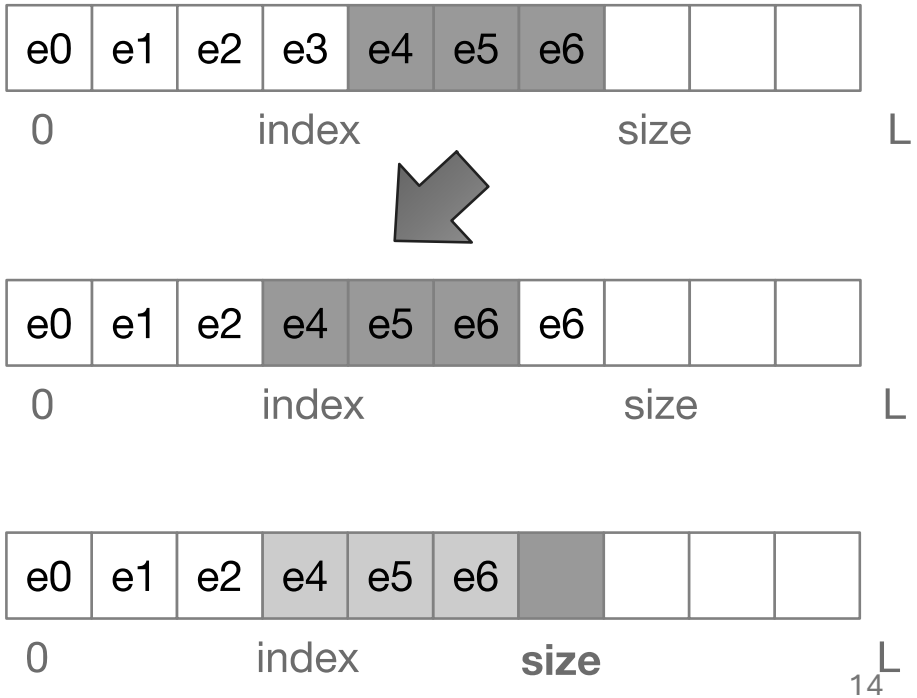
ArrayList

```
@Override
public void remove(int index) {
   checkIndexForAccess(index);
   moveSuffixOneLeft(index);
   elements[size - 1] = null;
   size -= 1;
}
```

```
private void moveSuffixOneLeft(int index) {
   // loop from left tp right (the suffix moves one to the left)
   // index is the destination
   for(int i = index; i < size - 1; i++) {
      elements[i] = elements[i + 1];
   }
}
```

```
elements[size - 1] = null;
size -= 1;
```

1. We check if the index is valid
2. We move the suffix [index+1, size) one position to the left
3. We remove the stalling reference to the last element of the list (for Garbage Collection)
4. We decrement the size



ArrayList

```
@Override
@SuppressWarnings("unchecked")
public E get(int index) {
   checkIndexForAccess(index);
   return (E) elements[index];
}
```

1. We check if the index is valid
2. We return the element at the index
   - We must do an unchecked cast
   - We suppress the warning

```
@Override
public int size() {
   return size;
}
```

1. We return the size of the list

```
@Override
public Iterator<E> iterator() {
   return new Itr();
}
```

1. An iterator is an object of type Iterator<E>
   - We create an instance of the class Itr
   - This class will implement the iterator interface for the ArrayList

ArrayList

```
public class MyArrayList<E> implements MyList<E> {

  // ...

  @Override
  public Iterator<E> iterator() {
    return new Itr();
  }

  private class Itr implements Iterator<E> {
    int cursor = 0;
    int lastReturned = -1;

    // ...

  }
}
```

- The Itr class is defined inside the ArrayList class
  - It is **not static**, so it's an **inner class**
  - As it is **private**, it can be accessed only from the ArrayList class
- Every instance of the inner class is **connected** to an instance of the outer class
  - **MyArrayList.this** is a reference to this instance (**qualified this**)
    - If the instance variable of the outer class is not hidden you can access it without it
  - Its constructor must be invoked
    - from an instance method of the outer class (uses this as outer instance)
    - or via an explicit reference to the outer instance

ArrayList

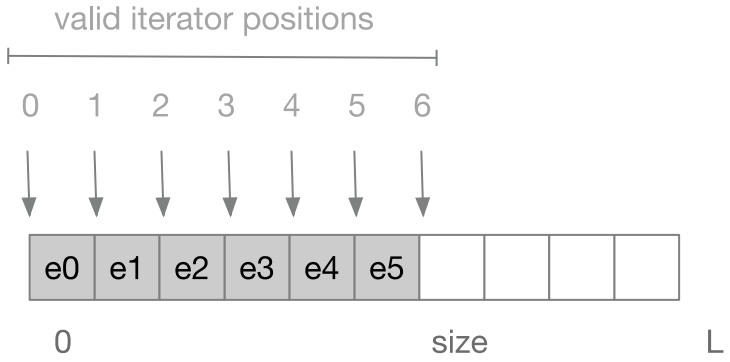
```
public class MyArrayList<E> implements MyList<E> {

  // ...

  @Override
  public Iterator<E> iterator() {
    return new Iterator<>() {
      int cursor = 0;
      int lastReturned = -1;

      // ...

    };
  }
}
```

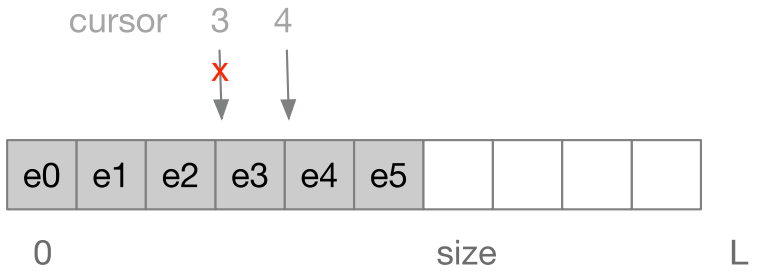
- In this case, we could have used an **anonymous inner class**
- In the same expression
  - We create an instance
  - Of an anonymous class
  - That implements Iterator<E>
    - As E can be deduced from the context, we use the diamond operator <>

ArrayList

```
private class Itr implements Iterator<E> {
    int cursor = 0;
    int lastReturned = -1;

    @Override
    public boolean hasNext() {
        return cursor < MyArrayList.this.size;
    }

    @Override
    public E next() {
        if (!hasNext()) throw new NoSuchElementException(
                "iterator has no more elements");
        @SuppressWarnings("unchecked")
        E next = (E) MyArrayList.this.elements[cursor];
        lastReturned = cursor;
        cursor += 1;
        return next;
    }
```

- The iterator has a cursor that refers to a position in the array (or beyond the last one)
- We have also the index of the last returned next (for remove)
- We have a next element if we are not beyond the last element



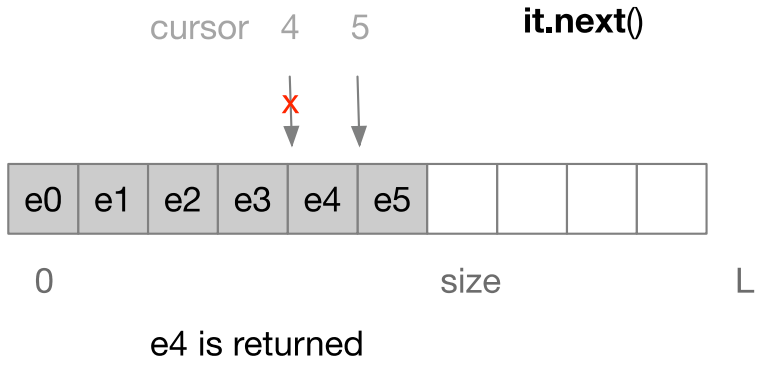


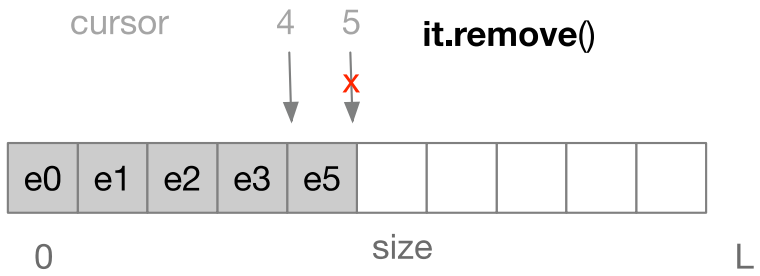
e3 is returned

ArrayList

```
@Override
  public void remove() {
     checkNextWasCalled();
     MyArrayList.this.remove(lastReturned);
     cursor -= 1;
     lastReturned = -1;
  }

  private void checkNextWasCalled() {
     if (lastReturned == -1)
        throw new IllegalStateException(
             "no next before remove");
  }
}
```

- The remove method deletes from the list the element returned by last
- It is an error to not have made a call to next before a call to remove
- NOTE: In this implementation we do not take into account co-modification, that is, that the list has been modified while using the iterator





ArrayList

```
@Override
public boolean equals(Object obj) {
   if (obj == this) return true;
   if (!(obj instanceof MyArrayList<?> other))
      return false;
   if (size != other. size) return false;
   for (int i = 0; i < size; i++) {
      if (!Objects.equals(elements[i], other.elements[i]))
         return false;
   }
   return true;
}
```

We use Objects.equals to avoid problems with null references

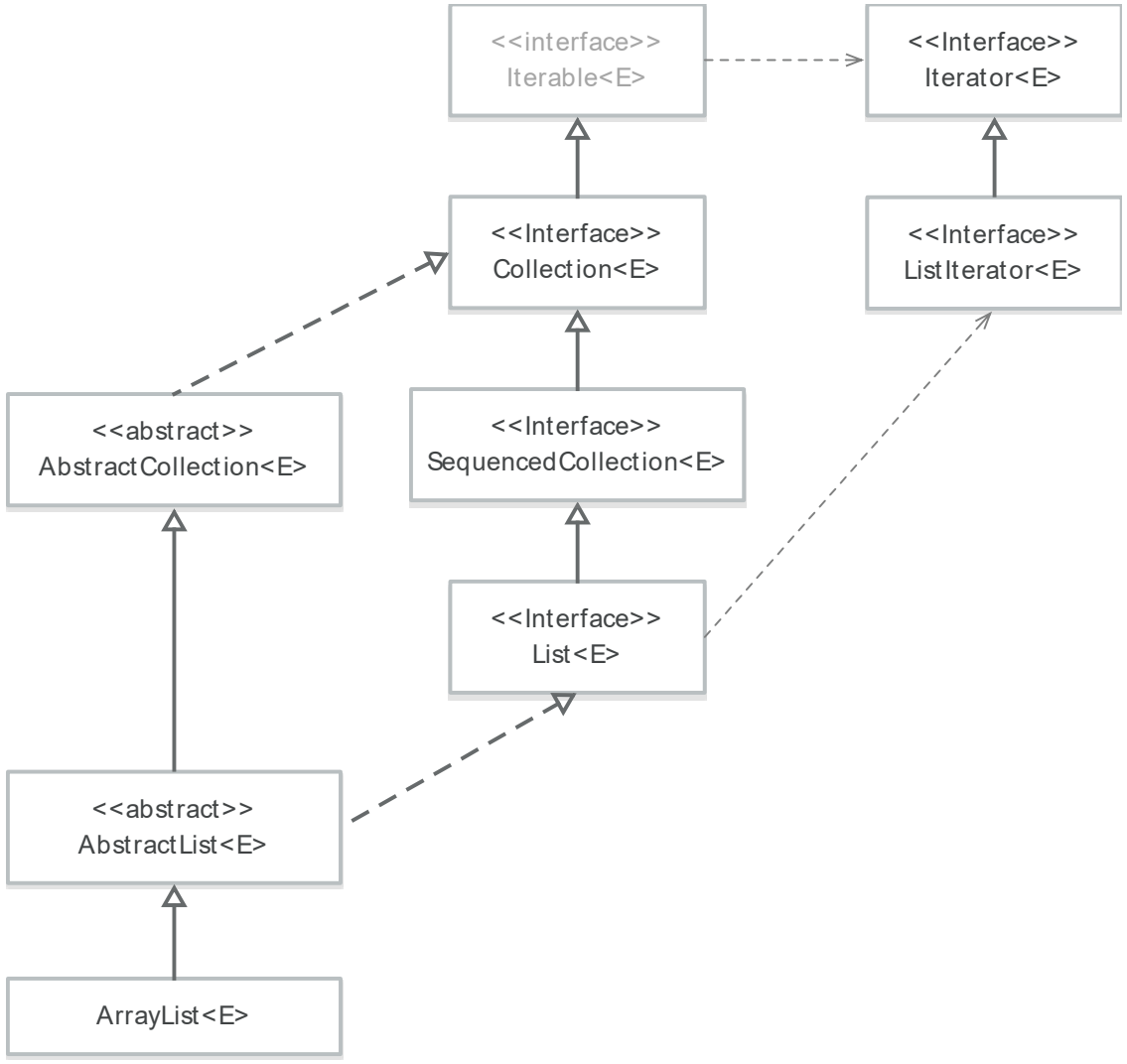
- Note the use of an unbounded wildcard in instanceof
  - Cannot use MyArrayList<E> due to erasure
- In this implementation we only allow to compare between array lists
  - In the JCF implementation equals works between different implementations of lists as well

ArrayList in the JCF

- Now, we have understood the idea behind implementing lists using extensible arrays
- The real implementation in the class ArrayList<E> is a little bit more complicated
  - The interface List<E> has much more methods
  - Not all the methods are implemented in the ArrayList<E> class
    - We have abstract superclasses that implement some of them
    - Why? Because they are inherited by other implementations of List<E>
- Now, we will present the overall structure of the classes and interfaces of the JCF relevant to the implementation of ArrayList<E>
- NOTE: For backwards compatibility, most of the implementation does not use some of the new elements of the language (e.g. default methods)

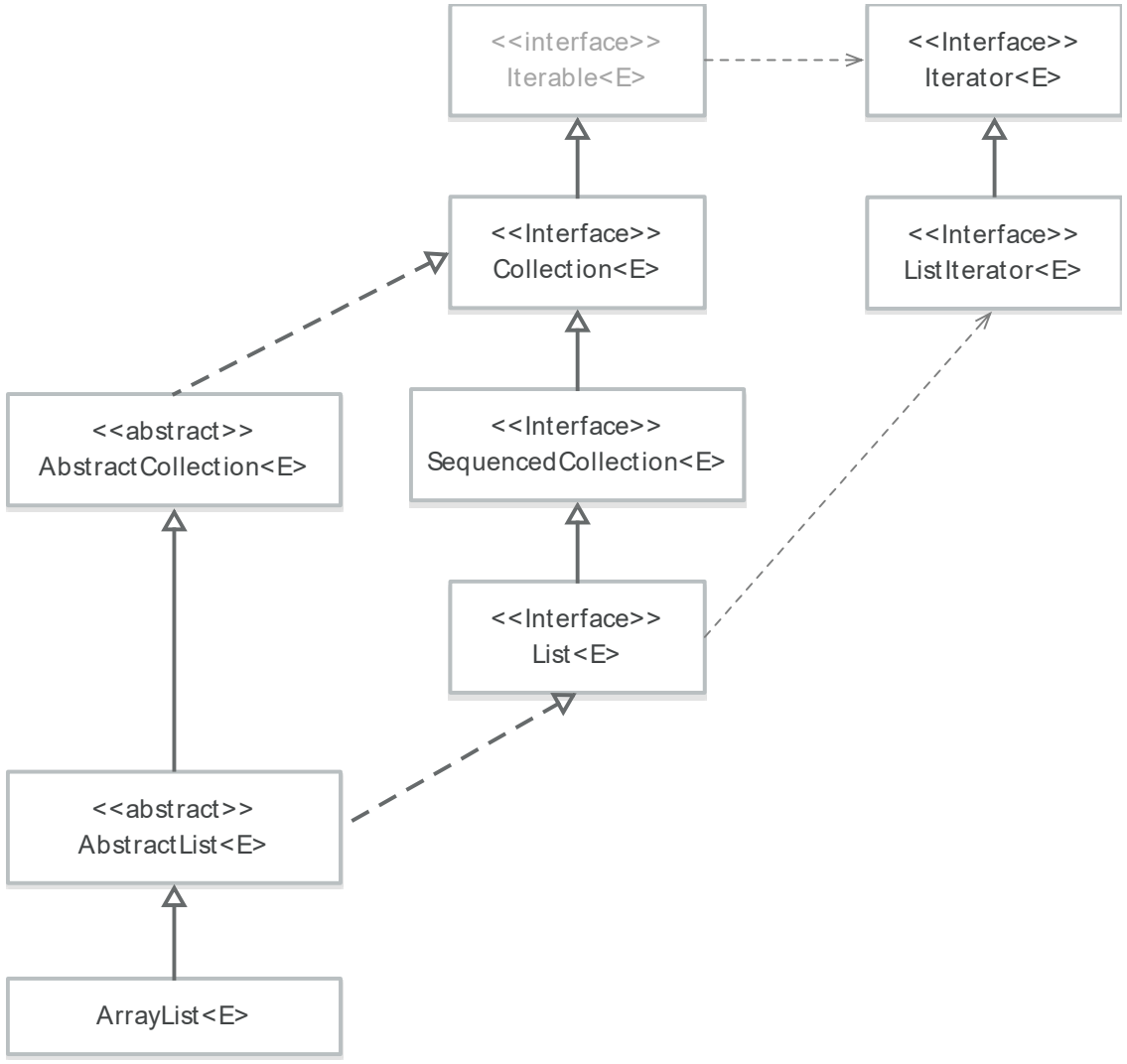
ArrayList in the JCF

- As we saw before, the interface List<E> gathers the methods declared in
  - Iterable<E>
  - Collection<E>
  - SequenceCollection<E>
- ListIterator<E>
  - It is an extension of the Iterator<E> interface
  - It allows moving from right to left
  - It allows modifying (not just removing) the element pointed by the iterator and adding new elements



ArrayList in the JCF

- AbstractCollection<E>
  - Implements some of the methods of Collection
  - By calling other implemented methods of AbstractCollection<E>
  - By calling abstract methods of Collection<E> that will be implemented by its subclasses
- AbstractList<E>
  - Uses a similar strategy for List<E>



Collection<E>

- **Represents a group of objects** known as its elements
  - Some collections allow duplicates, some don't
  - Some are ordered, some unordered
  - Some define encounter order, some don't
- Some **methods** are specified to be **optional**
  - They can throw **UnsupportedOperationException**
- **View** collections
  - Most collections manage storage of its elements, view collections rely on a backing collection
- **Unmodifiable collections**
  - A collection whose **mutator operations** throw **UnsupportedOperationException**
  - Any view collection backed by it must be unmodifiable as well
  - NOTE: They are not immutable because the elements can be mutable

Collection<E>

- boolean add(E e)
- boolean addAll(Collection<? extends E> c)
- void clear()
- boolean contains(Object a)
- boolean contains(Collection<?> c)
- boolean equals(Object o)
- int hashCode()
- boolean isEmpty()
- Iterator<E> iterator()
- default Stream<E> parallelStream()
- boolean remove(Object o)
- boolean removeAll(Collection<?> c)
- default boolean removeIf(Predicate<? super E> filter)
- boolean retainAll(Collection<?> c)
- int size()
- default SplitIterator<E> splitIterator()
- default Stream<E> stream()
- Object[] toArray()
- default <T> T[] toArray(IntFunction<T[]> generator)
- <T> T[] toArray(T[] a)

AbstractCollection<E>

- It provides a skeletal implementation of Collection<E> to minimize the effort to implement it
- **Unmodifiable** collection
  - **iterator** and **size**
- **Modifiable** collection needs also
  - **add**
  - **remove** operation **in** the Iterator<E> returned by **iterator**
- Its subclasses should generally provide also
  - No argument and Collection constructor
- The **specification** of each non-abstract method **describe in detail its implementation**
  - It may be overridden if the collection admits a more efficient implementation

AbstractCollection<E>

- We will show the implementation of some of the methods
- But it is advisable to read the code for all of them
  - Those marked in grey in the previous list are skippable (they are about streams and/or use lambdas, which go beyond this course)
- **Reading code is one of the most important capabilities a programmer must acquire !!**
  - And the code of the JCF is (with exceptions) a good way to learn the style of programming required to implement data structures
- How to access the implementation:
  - From IntelliJ, Shift Shift and ask for the class (e.g. java.util.ArrayList)
    - Use a project configured with Java 21 (last LTS)
  - Open JDK GitHub repository

AbstractCollection<E>

```
public abstract class AbstractCollection<E> implements Collection<E> {

  protected AbstractCollection() { }

  public boolean isEmpty() {
    return size() == 0;
  }

  public boolean contains(Object o) {
    Iterator<E> it = iterator();
    if (o==null) {
    while (it.hasNext())
      if (it.next()==null)
        return true;
    } else {
      while (it.hasNext())
        if (o.equals(it.next()))
          return true;
    }
    return false;
  }

  public boolean add(E e) {
    throw new UnsupportedOperationException();
  }

  public boolean remove(Object o) {
    Iterator<E> it = iterator();
    if (o==null) {
      while (it.hasNext()) {
        if (it.next()==null) {
          it.remove();
          return true;
        }
      }
    } else {
      while (it.hasNext()) {
        if (o.equals(it.next())) {
          it.remove();
          return true;
        }
      }
    }
    return false;
  }
```

AbstractCollection<E>

```
public boolean containsAll(Collection<?> c) {
   for (Object e : c)
      if (!contains(e))
         return false;
   return true;
}

public boolean addAll(Collection<? extends E> c) {
   boolean modified = false;
   for (E e : c)
      if (add(e))
         modified = true;
   return modified;
}
```

```
public boolean removeAll(Collection<?> c) {
   Objects.requireNonNull(c);
   boolean modified = false;
   Iterator<?> it = iterator();
   while (it.hasNext()) {
      if (c.contains(it.next())) {
         it.remove();
         modified = true;
      }
   }
   return modified;
}

public void clear() {
   Iterator<E> it = iterator();
   while (it.hasNext()) {
      it.next();
      it.remove();
   }
}
```

AbstractCollection<E>

```
public boolean retainAll(Collection<?> c) {
    Objects.requireNonNull(c);
    boolean modified = false;
    Iterator<E> it = iterator();
    while (it.hasNext()) {
        if (!c.contains(it.next())) {
            it.remove();
            modified = true;
        }
    }
    return modified;
}
```

```
public String toString() {
    Iterator<E> it = iterator();
    if (! it.hasNext())
        return "[]";

    StringBuilder sb = new StringBuilder();
    sb.append('[');
    for (;;) {
        E e = it.next();
        sb.append(e == this ? "(this Collection)" : e);
        if (! it.hasNext())
            return sb.append(']').toString();
        sb.append(',').append(' ');
    }
}

// toArray method omitted for mental sanity ☺
```

SequencedCollection<E>

- A collection that
  - has a well-defined encounter order
    - Conceptually, they have a linear arrangement from first to last
  - supports operations at both ends
  - is reversible
- Mostly implemented by default methods
  - It was added in Java 21
  - Without breaking any existing code !!

SequencedCollection<E>

```
public interface SequencedCollection<E> extends Collection<E> {

  SequencedCollection<E> reversed();

  default void addFirst(E e) {
    throw new UnsupportedOperationException();
  }

  default void addLast(E e) {
    throw new UnsupportedOperationException();
  }

  default E getFirst() {
    return this.iterator().next();
  }

  default E getLast() {
      return this.reversed().iterator().next();
    }

  default E removeFirst() {
    var it = this.iterator();
    E e = it.next();
    it.remove();
    return e;
  }

  default E removeLast() {
    var it = this.reversed().iterator();
    E e = it.next();
    it.remove();
    return e;
  }
}
```

List<E>

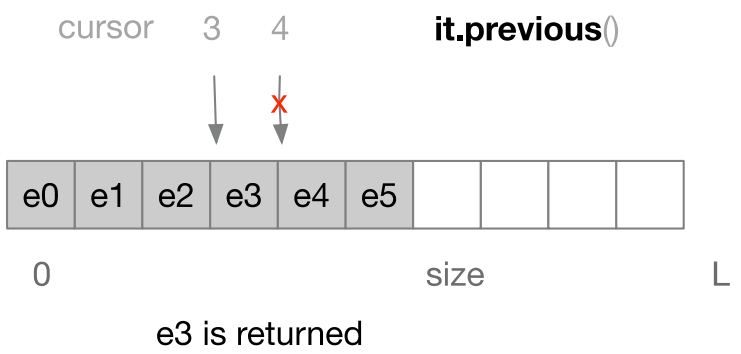
- An ordered (by position) collection
  - The user has precise control of where each element is added
  - The user can access elements by position
- Typically allows duplicates and null elements
- Refines the specification of some methods
  - iterator, add, remove, equals, hashCode
- Adds methods for indexed access (with int parameter for index)
  - add, addAll, get, remove, set
- Provides for a special iterator called ListIterator<E>
- Methods for search an element
  - indexOf, lastIndexOf
- Unmodifiable Lists
  - Static methods List.of, List.copyOf to create unmodifiable lists

Iterator<E>

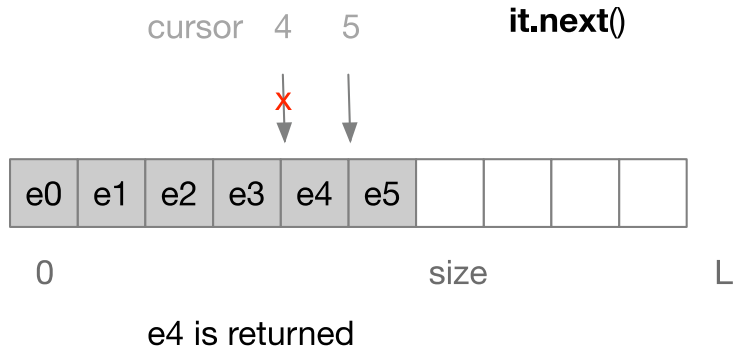
- In our toy implementation of lists, we have not considered what happens when one modifies the list while using an iterator
  - It will fail with a ConcurrentModificationException when detects this situation
  - To do so,
    - the list has a modCount attribute that is incremented when modifying the list
    - when creating an iterator, the current value is saved
    - the operations in the iterator check if the saved value is still the same
    - NOTE: It is defined as transient to not include it in the serialization of the list

ListIterator<E>

- An extension of Iterator<E> that allows
  - Traversing the list in each direction
  - Modifying the list while iterating
  - Obtaining the iterator's "current" position in the list
- A ListIterator<E> has no "current" position in the list
  - Conceptually, it is between the element that will be returned by previous and the element that will be returned by next



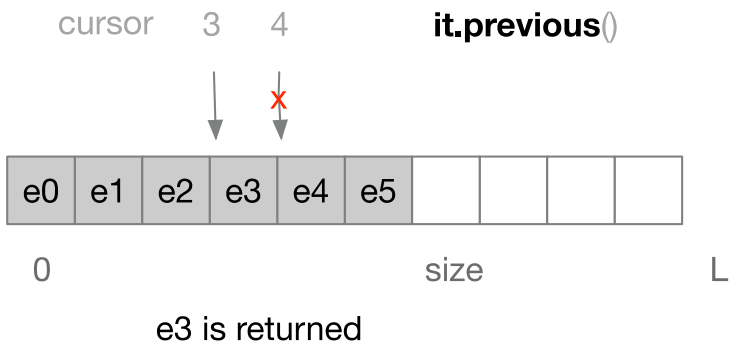
e3 is returned

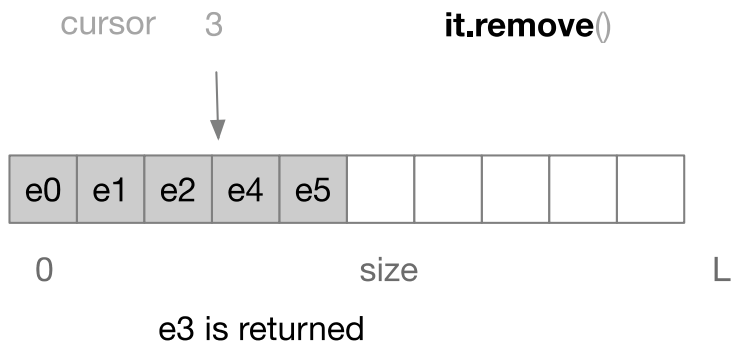


e4 is returned

ListIterator<E>

- You can also add at “the position” of the iterator
- And you can modify the value at the position returned by previous / next
- Obviously, the remove operator also works for the last returned previous element





AbstractList<E>

- It provides a **skeletal implementation of List<E>** to minimize the effort to implement it
- Your implementation must be **backed by a "random access" data store** (such as an array)
  - For a "sequential access" implementation, use AbstractSequentialList<E>
- To implement an **unmodifiable** list
  - **get(int)** and **size**
- To implement a **modifiable** list
  - **set(int, E)**
  - If it's **variable size**: **add(int, E)** and **remove(int)**
- The **specification** of each non-abstract method **describes its implementation**
  - Meant to be extended
  - Override them if the collection allows for a more efficient implementations (e.g. ArrayList<E> overrides all of them)

AbstractList<E>

```
public abstract class AbstractList<E>
    extends AbstractCollection<E> implements List<E> {

  protected transient int modCount = 0;

  protected AbstractList() { }

  public boolean add(E e) {
    add(size(), e);
    return true;
  }

  public abstract E get(int index);

  public E set(int index, E element) {
    throw new UnsupportedOperationException();
  }

  public void add(int index, E element) {
    throw new UnsupportedOperationException();
  }

  public E remove(int index) {
    throw new UnsupportedOperationException();
  }
```

```
  public int indexOf(Object o) {
    ListIterator<E> it = listIterator();
    if (o==null) {
      while (it.hasNext())
        if (it.next()==null)
          return it.previousIndex();
    } else {
      while (it.hasNext())
        if (o.equals(it.next()))
          return it.previousIndex();
    }
    return -1;
  }

  public int lastIndexOf(Object o) {
    ListIterator<E> it = listIterator(size());
    if (o==null) {
      while (it.hasPrevious())
        if (it.previous()==null)
          return it.nextIndex();
    } else {
      while (it.hasPrevious())
        if (o.equals(it.previous()))
          return it.nextIndex();
    }
    return -1;
  }
```

AbstractList<E> (Iterator<E>)

```
public Iterator<E> iterator() { return new Itr(); }

private class Itr implements Iterator<E> {
  int cursor = 0;
  int lastRet = -1;
  int expectedModCount = modCount;

  public boolean hasNext() {
    return cursor != size();
  }

  public E next() {
    checkForComodification();
    try {
      int i = cursor;
      E next = get(i);
      lastRet = i;
      cursor = i + 1;
      return next;
    } catch (IndexOutOfBoundsException e) {
      checkForComodification();
      throw new NoSuchElementException(e);
    }
  }

  public void remove() {
    if (lastRet < 0)
      throw new IllegalStateException();
    checkForComodification();

    try {
      AbstractList.this.remove(lastRet);
      if (lastRet < cursor)
        cursor--;
      lastRet = -1;
      expectedModCount = modCount;
    } catch (IndexOutOfBoundsException e) {
      throw new ConcurrentModificationException();
    }
  }

  final void checkForComodification() {
    if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
  }
}
```

AbstractList<E> (ListIterator<E>)

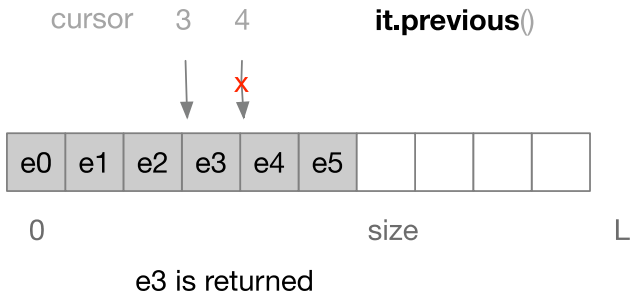
```
public ListIterator<E> listIterator() {
   return listIterator(0);
}

public ListIterator<E> listIterator(final int index) {
   rangeCheckForAdd(index);
   return new ListItr(index);
}

private class ListItr extends Itr implements ListIterator<E> {
   ListItr(int index) {
      cursor = index;
   }

   public boolean hasPrevious() {
      return cursor != 0;
   }

   public int nextIndex() {
      return cursor;
   }

   public int previousIndex() {
      return cursor-1;
   }
```

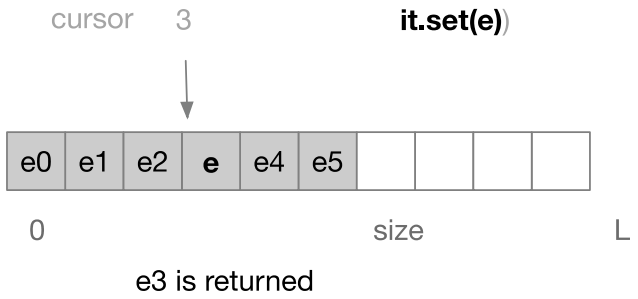
```
   public E previous() {
      checkForComodification();
      try {
         int i = cursor - 1;
         E previous = get(i);
         lastRet = cursor = i;
         return previous;
      } catch (IndexOutOfBoundsException e) {
         checkForComodification();
         throw new NoSuchElementException(e);
      }
   }
```

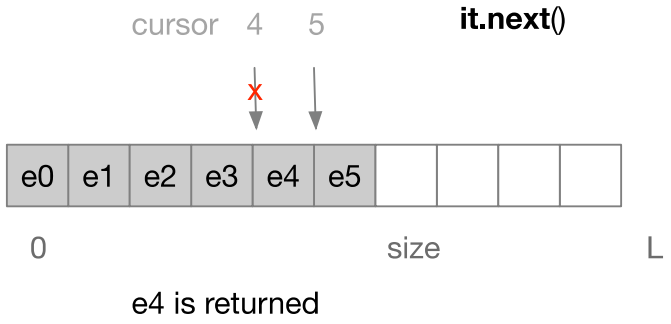
AbstractList<E> (ListIterator<E>)

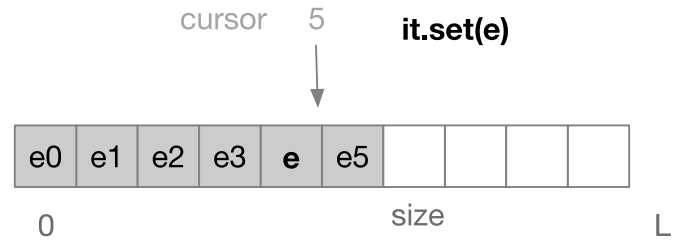
```
public void set(E e) {
    if (lastRet < 0)
      throw new IllegalStateException();
    checkForComodification();

    try {
      AbstractList.this.set(lastRet, e);
      expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
      throw new ConcurrentModificationException();
    }
  }
```

- Replaces the last element returned by previous / next
- If can be made when neither remove nor add has been called after the last call of previous / next





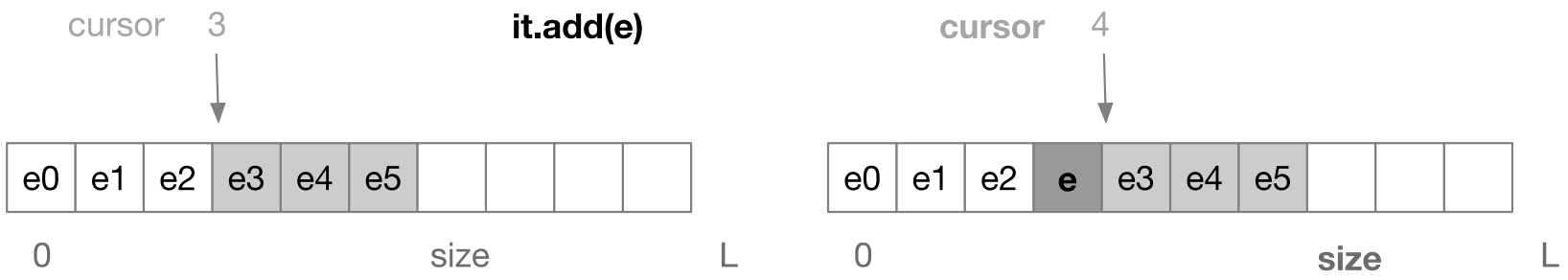




AbstractList<E> (ListIterator<E>)

```
public void add(E e) {
    checkForComodification();

    try {
        int i = cursor;
        AbstractList.this.add(i, e);
        lastRet = -1;
        cursor = i + 1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
  }
}
```

- The element is inserted
  - before the element that would be returned by next (if any)
  - after the element that would be returned by previous (if any)
- A subsequent call to next wouldn't be affected
- A subsequent call to previous returns the new element



ArrayList<E>

- Resizable-array implementation of List<E>
- Implements all optional operations and permits null elements
- Besides implementing List<E>, it provides some methods to manipulate the underlying array
- It has a capacity, which is the size of the underlying array that is always at least equal to the size of the list
  - The growth policy only requires amortized constant time for add
- The implementation is not synchronized
- Iterators are fail-fast:
  - Fail if a structural modification of the list is detected after creating the iterator
  - Do not guarantee correctness, only detects some bugs

ArrayList<E>

```
public class ArrayList<E> extends AbstractList<E>
      implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

   @java.io.Serial
   private static final long serialVersionUID = 8683452581122892189L;

   private static final int DEFAULT_CAPACITY = 10;

   private static final Object[] EMPTY_ELEMENTDATA = {};

   private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

   transient Object[] elementData;

   private int size;

  // ....
}
```

- **RandomAccess**: marker that indicates constant time random access
- **Cloneable**: cloning machinery
- **Serializable**: serializable machinery

- **DEFAULT_CAPACITY**: default capacity of the list
- **EMPTY_ELEMENTDATA**: shared array instance used for empty instances
- **DEFAULTCAPACITY_EMPTY_ELEMENTDATA**: shared empty array used for default sized empty instances (distinguished from the previous to know how much inflate on first insertion)

ArrayList<E>

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

public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

```
public ArrayList(Collection<? extends E> c) {
    Object[] a = c.toArray();
    if ((size = a.length) != 0) {
        if (c.getClass() == ArrayList.class) {
            elementData = a;
        } else {
            elementData = Arrays.copyOf(a, size, Object[].class);
        }
    } else {
        // replace with empty array.
        elementData = EMPTY_ELEMENTDATA;
    }
}
```

- ArrayLists without capacity are **initialized lazily** (on first insertion)
- As we do not have the guarantee that the array returned by toArray can be aliased we do a **defensive copy**
- As we know its implementation in ArrayList, and we know it's a fresh copy, we do not do a defensive copy in that case

ArrayList<E>

```
public int size() {
   return size;
}

public boolean isEmpty() {
   return size == 0;
}
```

```
public void trimToSize() {
   modCount++;
   if (size < elementData.length) {
      elementData = (size == 0)
       ? EMPTY_ELEMENTDATA
       : Arrays.copyOf(elementData, size);
   }
}
```

```
private Object[] grow(int minCapacity) {
   int oldCapacity = elementData.length;
   if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
      int newCapacity = ArraysSupport.newLength(oldCapacity,
            minCapacity - oldCapacity, /* minimum growth */
            oldCapacity >> 1           /* preferred growth */);
      return elementData = Arrays.copyOf(elementData, newCapacity);
   } else {
      return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];
   }
}

private Object[] grow() {
   return grow(size + 1);
}
```

```
public void ensureCapacity(int minCapacity) {
   if (minCapacity > elementData.length
      && !(elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
         && minCapacity <= DEFAULT_CAPACITY)) {
      modCount++;
      grow(minCapacity);
   }
}
```

ArraySupport

- in package jdk.internal.util
- not accessible out of JDK

```
public class ArraysSupport {

  // ...

  public static final int SOFT_MAX_ARRAY_LENGTH = Integer.MAX_VALUE - 8;

  public static int newLength(int oldLength, int minGrowth, int prefGrowth) {
    int prefLength = oldLength + Math.max(minGrowth, prefGrowth); // might overflow
    if (0 < prefLength && prefLength <= SOFT_MAX_ARRAY_LENGTH) {
      return prefLength;
    } else {
      return hugeLength(oldLength, minGrowth);
    }
  }

  private static int hugeLength(int oldLength, int minGrowth) {
    int minLength = oldLength + minGrowth;
    if (minLength < 0) { // overflow
      throw new OutOfMemoryError(
          "Required array length " + oldLength + " + " + minGrowth + " is too large");
    } else if (minLength <= SOFT_MAX_ARRAY_LENGTH) {
      return SOFT_MAX_ARRAY_LENGTH;
    } else {
      return minLength;
    }
  }
}
```

ArrayList<E>

```
private void add(E e, Object[] elementData, int s) {
  if (s == elementData.length)
    elementData = grow();
  elementData[s] = e;
  size = s + 1;
}

public boolean add(E e) {
  modCount++;
  add(e, elementData, size);
  return true;
}


public void addFirst(E element) {
  add(0, element);
}
```

```
public void add(int index, E element) {
  rangeCheckForAdd(index);
  modCount++;
  final int s;
  Object[] elementData;
  if ((s = size) == (elementData = this.elementData).length)
    elementData = grow();
  System.arraycopy(elementData, index,
                   elementData, index + 1,
                   s - index);
  elementData[index] = element;
  size = s + 1;
}


public void addLast(E element) {
  add(element);
}
```

ArrayList<E>

```
@SuppressWarnings("unchecked")
E elementData(int index) {
  return (E) elementData[index];
}

@SuppressWarnings("unchecked")
static <E> E elementAt(Object[] es, int index) {
  return (E) es[index];
}

public E get(int index) {
  Objects.checkIndex(index, size);
  return elementData(index);
}

public E getFirst() {
  if (size == 0) {
    throw new NoSuchElementException();
  } else {
    return elementData(0);
  }
}
```

```
public E getLast() {
  int last = size - 1;
  if (last < 0) {
    throw new NoSuchElementException();
  } else {
    return elementData(last);
  }
}

public E set(int index, E element) {
  Objects.checkIndex(index, size);
  E oldValue = elementData(index);
  elementData[index] = element;
  return oldValue;
}
```

ArrayList<E>

```
public E remove(int index) {
    Objects.checkIndex(index, size);
    final Object[] es = elementData;
    @SuppressWarnings("unchecked") E oldValue = (E) es[index];
    fastRemove(es, index);
    return oldValue;
}


public E removeFirst() {
    if (size == 0) {
        throw new NoSuchElementException();
    } else {
        Object[] es = elementData;
        @SuppressWarnings("unchecked") E oldValue = (E) es[0];
        fastRemove(es, 0);
        return oldValue;
    }
}
```

```
public E removeLast() {
    int last = size - 1;
    if (last < 0) {
        throw new NoSuchElementException();
    } else {
        Object[] es = elementData;
        @SuppressWarnings("unchecked") E oldValue = (E) es[last];
        fastRemove(es, last);
        return oldValue;
    }
}


private void fastRemove(Object[] es, int i) {
    modCount++;
    final int newSize;
    if ((newSize = size - 1) > i)
        System.arraycopy(es, i + 1, es, i, newSize - i);
    es[size = newSize] = null;
}
```

ArrayList<E> (Iterator<E>)

```
/**
 * An optimized version of AbstractList.Itr
 */
private class Itr implements Iterator<E> {
    int cursor;       // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;

    // prevent creating a synthetic constructor
    Itr() {}

    public boolean hasNext() {
        return cursor != size;
    }
```

```
    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }
```

```
    public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();
        try {
            ArrayList.this.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
```

ArrayList<E> (ListIterator<E>)

```
/**
 * An optimized version of AbstractList.ListItr
 */
private class ListItr extends Itr implements ListIterator<E> {
    ListItr(int index) {
        super();
        cursor = index;
    }

    public boolean hasPrevious() {
        return cursor != 0;
    }

    public int nextIndex() {
        return cursor;
    }

    public int previousIndex() {
        return cursor - 1;
    }

    @SuppressWarnings("unchecked")
    public E previous() {
        checkForComodification();
        int i = cursor - 1;
        if (i < 0)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i;
        return (E) elementData[lastRet = i];
    }
```

ArrayList<E> (ListIterator<E>)

```
public void set(E e) {
   if (lastRet < 0)
      throw new IllegalStateException();
   checkForComodification();

   try {
      ArrayList.this.set(lastRet, e);
   } catch (IndexOutOfBoundsException ex) {
      throw new ConcurrentModificationException();
   }
}
```

```
public void add(E e) {
      checkForComodification();

      try {
         int i = cursor;
         ArrayList.this.add(i, e);
         cursor = i + 1;
         lastRet = -1;
         expectedModCount = modCount;
      } catch (IndexOutOfBoundsException ex) {
         throw new ConcurrentModificationException();
      }
   }
}
```

ArrayList<E> and AbstractList<E>

- Both implement, the former totally and the later partially, List<E> with **data baked by an efficient random-access store**:
  - ArrayList<E> uses an array
  - AbstractList<E> uses efficient get / set / add / remove by index
- But, what if the implementation of List<E> is backed by a **sequence of linked nodes**?
  - So that **we do not have efficient access by index**
- The classes AbstractSequentialList<E> and LinkedList<E> provides us with a partial and a total implementation of this design

AbstractSequentialList<E>

- It's an extension of AbstractList<E> that provides a skeletal implementation of the List<E> interface by a "sequential access" data store
- This class is the opposite of AbstractList<E> in the sense that the random-access operations get / set / add / remove are implemented on top of the ListIterator<E> returned by listIterator(int)
  - in AbstractList<E> it is the other way around
- To implement a List<E> the programmer must implement
  - **size** and **listIterator(int)** with **next**, **previous**, **hasXXX**, and **XXXIndex**
  - For **modifiable** list, iterator's **set(E)**
  - For **variable size**, iterator's **add(E)** and **remove**

AbstractSequentialList<E>

```
public abstract class AbstractSequentialList<E>
      extends AbstractList<E> {

   protected AbstractSequentialList() { }

   public E get(int index) {
      try {
         return listIterator(index).next();
      } catch (NoSuchElementException exc) {
         throw new IndexOutOfBoundsException("Index:"+index);
      }
   }

   public E set(int index, E element) {
      try {
         ListIterator<E> e = listIterator(index);
         E oldVal = e.next();
         e.set(element);
         return oldVal;
      } catch (NoSuchElementException exc) {
         throw new IndexOutOfBoundsException("Index: "+index);
      }
}
```

```
   public void add(int index, E element) {
      try {
         listIterator(index).add(element);
      } catch (NoSuchElementException exc) {
         throw new IndexOutOfBoundsException("Index: "+index);
      }
   }

   public E remove(int index) {
      try {
         ListIterator<E> e = listIterator(index);
         E outCast = e.next();
         e.remove();
         return outCast;
      } catch (NoSuchElementException exc) {
         throw new IndexOutOfBoundsException("Index: "+index);
      }
   }
```

- **as we'll see, the creation of a listIterator at a given index is linear, so all these indexed operations will be linear as well**

AbstractSequentialList<E>

```
public boolean addAll(int index, Collection<? extends E> c) {
      try {
         boolean modified = false;
         ListIterator<E> e1 = listIterator(index);
         for (E e : c) {
            e1.add(e);
            modified = true;
         }
         return modified;
      } catch (NoSuchElementException exc) {
         throw new IndexOutOfBoundsException("Index: "+index);
      }
   }

   public Iterator<E> iterator() {
      return listIterator();
   }

   public abstract ListIterator<E> listIterator(int index);
}
```

- **All these random-access operations by index do a sequential search**
- *listIterator(int)* is a **concrete** operation in AbstractList<E>, but here we mark it as **abstract**
  - we **do not inherit** its base class implementation
  - and our implementors **must also implement it** !!

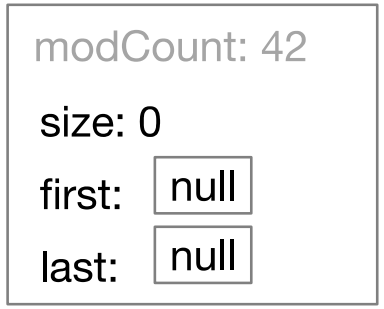
LinkedList<E>

- Doubly-linked implementation of the List<E> and Deque<E> interfaces
- Implements all optional operations and permits null elements
- All the operations perform as one expects them for a doubly-linked list
  - Index operations perform a traverse from the closest end
- The implementation is not synchronized
- Iterators are fail-fast:
  - They fail if a structural modification of the list is detected after creating the iterator
  - And they do not guarantee correctness, only detects some bugs

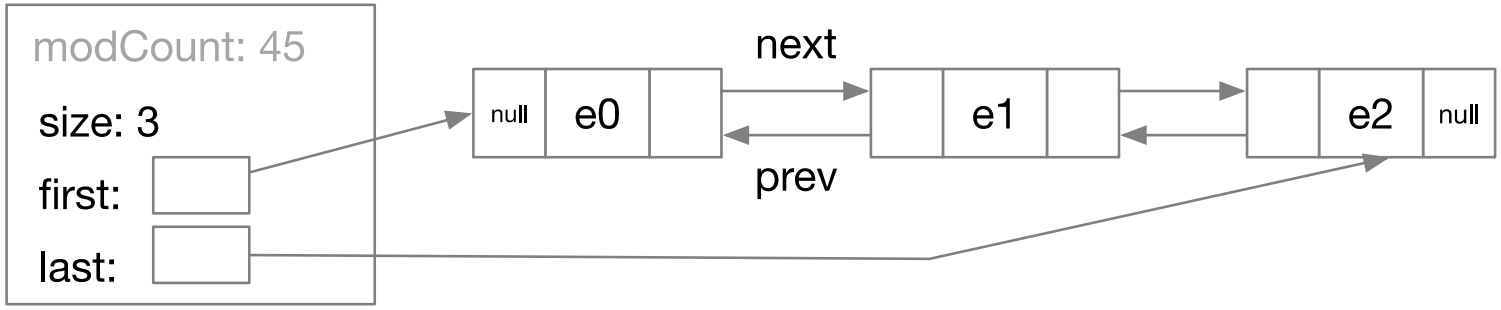
LinkedList<E>

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, java.io.Serializable {

  transient int size = 0;

  /**
   * Pointer to first node.
   */
  transient Node<E> first;

  /**
   * Pointer to last node.
   */
  transient Node<E> last;

  /*
  void dataStructureInvariants() {
     assert (size == 0)
        ? (first == null && last == null)
        : (first.prev == null && last.next == null);
  }
  */
```

Empty list:



A list with three elements:



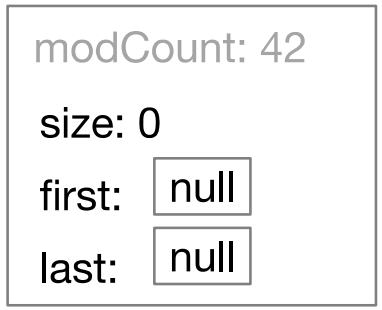
LinkedList<E> (Node<E>)

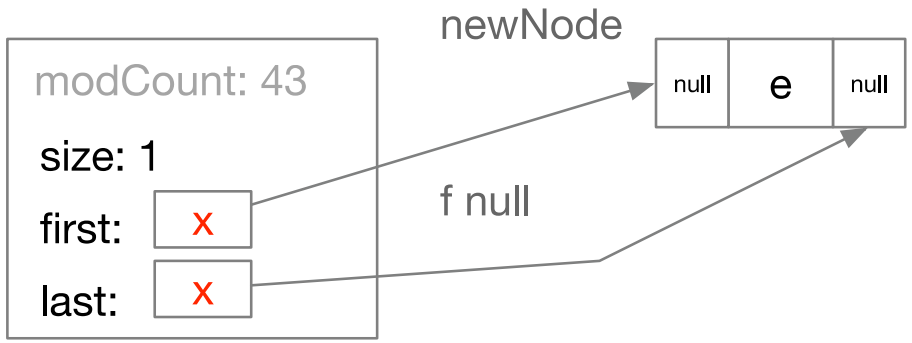
```
private static class Node<E> {
   E item;
   Node<E> next;
   Node<E> prev;

   Node(Node<E> prev, E element, Node<E> next) {
      this.item = element;
      this.next = next;
      this.prev = prev;
   }
}
```

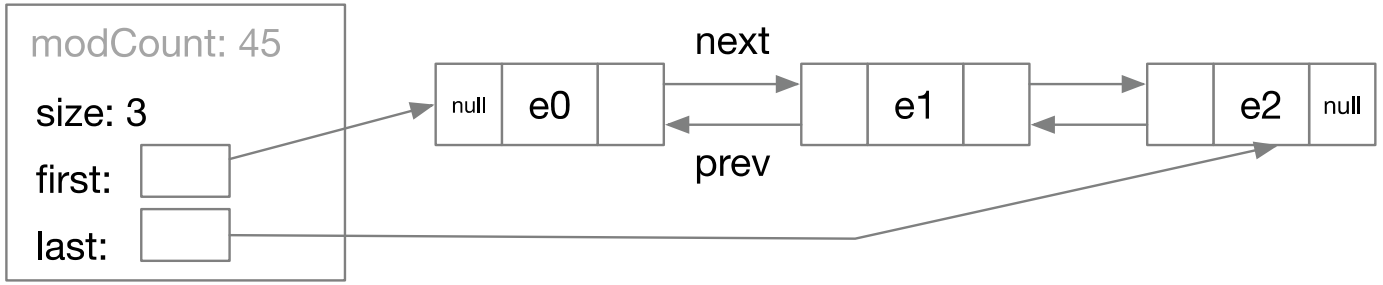
- A Node<E> is implemented by a **static nested class**
  - It has no access to the attributes of the LinkedList (no qualified this)
  - The parameter <E> is different from that of the LinkedList<E>
- It's a **doubly-linked list** because each Node<E> has a references to
  - Its **previous** node in the list
  - Its **next** node in the list

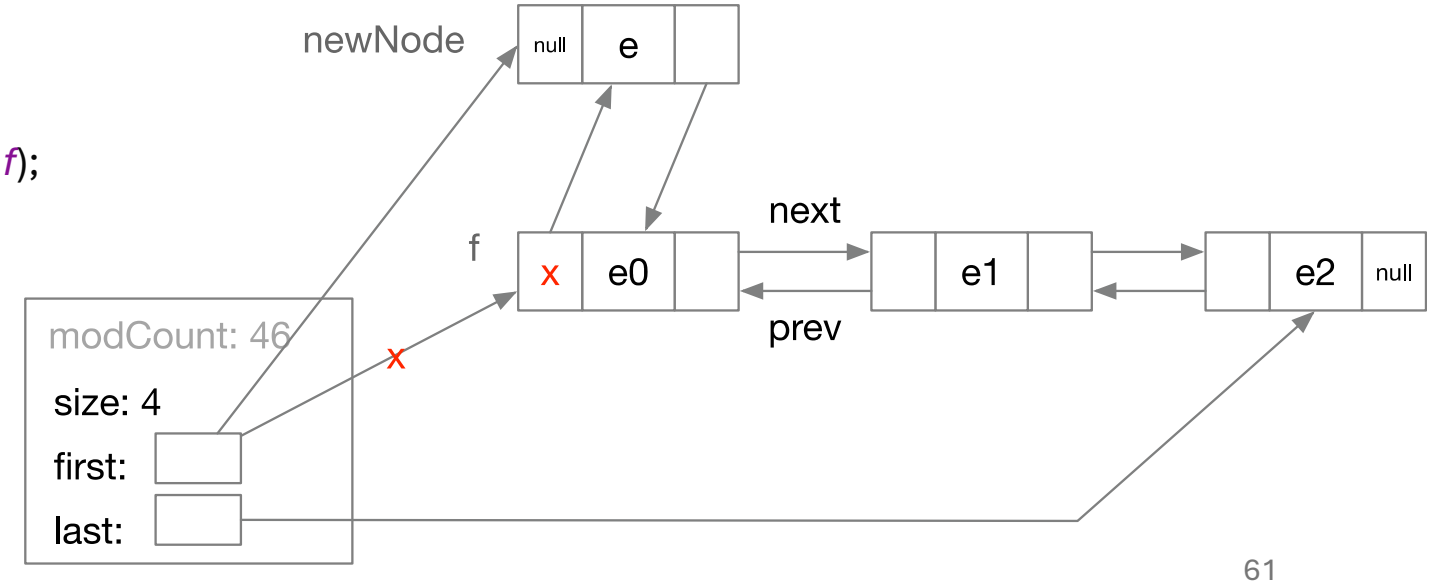
LinkedList<E>

```
public LinkedList() { }

public LinkedList(Collection<? extends E> c) {
   this();
   addAll(c);
}

/**
 * Links e as first element.
 */
private void linkFirst(E e) {
   final Node<E> f = first;
   final Node<E> newNode = new Node<>(null, e, f);
   first = newNode;
   if (f == null)
      last = newNode;
   else
      f.prev = newNode;
   size++;
   modCount++;
}
```









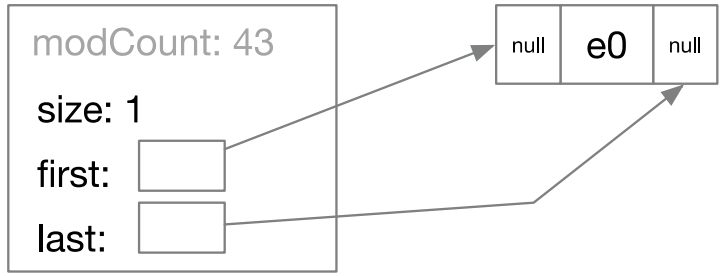
LinkedList<E>

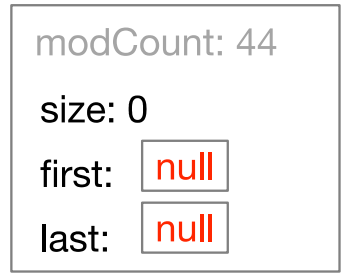
```
/**
 * Links e as last element.
 */
void linkLast(E e) {
   final Node<E> l = last;
   final Node<E> newNode = new Node<>(l, e, null);
   last = newNode;
   if (l == null)
      first = newNode;
   else
      l.next = newNode;
   size++;
   modCount++;
}
```

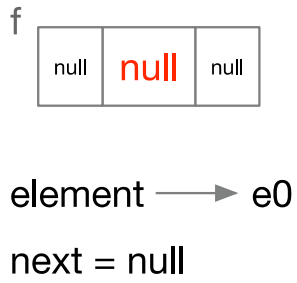
```
/**
 * Inserts element e before non-null Node succ.
 */
void linkBefore(E e, Node<E> succ) {
   // assert succ != null;
   final Node<E> pred = succ.prev;
   final Node<E> newNode = new Node<>(pred, e, succ);
   succ.prev = newNode;
   if (pred == null)
      first = newNode;
   else
      pred.next = newNode;
   size++;
   modCount++;
}
```

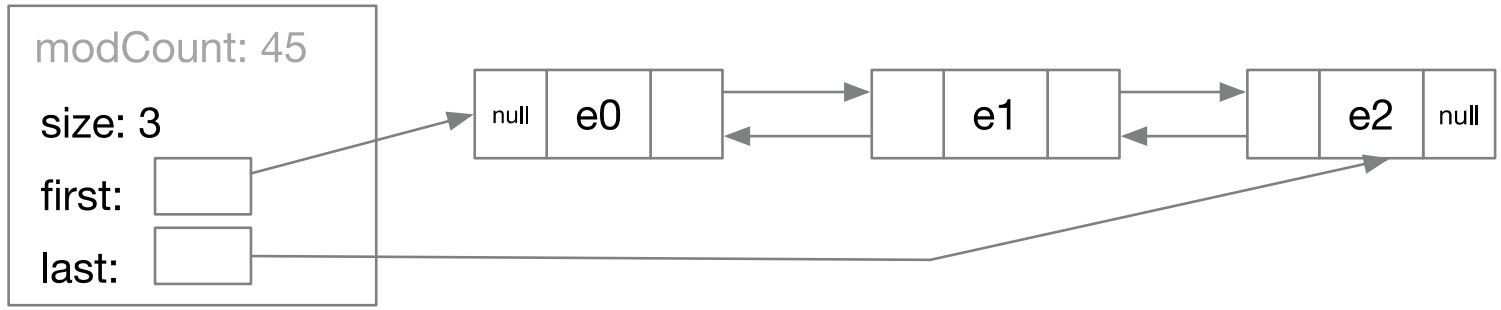
LinkedList<E>

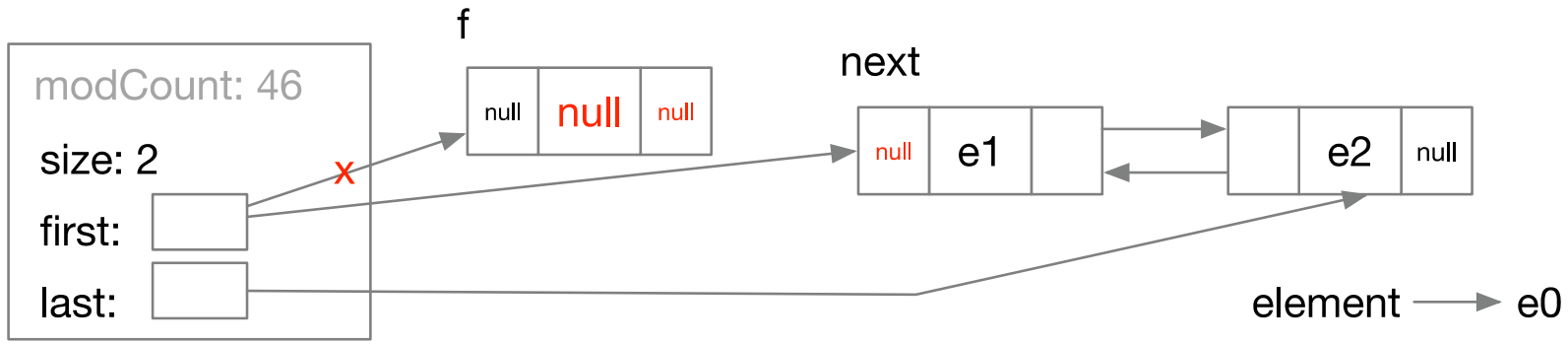
```
/**
 * Unlinks non-null first node f.
 */
private E unlinkFirst(Node<E> f) {
   // assert f == first && f != null;
   final E element = f.item;
   final Node<E> next = f.next;
   f.item = null;
   f.next = null; // help GC
   first = next;
   if (next == null)
      last = null;
   else
      next.prev = null;
   size--;
   modCount++;
   return element;
}
```











LinkedList<E>

```
/**
 * Unlinks non-null last node l.
 */
private E unlinkLast(Node<E> l) {
    // assert l == last && l != null;
    final E element = l.item;
    final Node<E> prev = l.prev;
    l.item = null;
    l.prev = null; // help GC
    last = prev;
    if (prev == null)
        first = null;
    else
        prev.next = null;
    size--;
    modCount++;
    return element;
}
```

```
/**
 * Unlinks non-null node x.
 */
E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;
    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }
    if (next == null) {
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
    }
    x.item = null;
    size--;
    modCount++;
    return element;
}
```

LinkedList<E>

```
public E getFirst() {
   final Node<E> f = first;
   if (f == null)
      throw new NoSuchElementException();
   return f.item;
}

public E getLast() {
   final Node<E> l = last;
   if (l == null)
      throw new NoSuchElementException();
   return l.item;
}

public E removeFirst() {
   final Node<E> f = first;
   if (f == null)
      throw new NoSuchElementException();
   return unlinkFirst(f);
}
```

```
public E removeLast() {
   final Node<E> l = last;
   if (l == null)
      throw new NoSuchElementException();
   return unlinkLast(l);
}

public void addFirst(E e) {
   linkFirst(e);
}

public void addLast(E e) {
   linkLast(e);
}

public boolean contains(Object o) {
   return indexOf(o) >= 0;
}

public int size() {
   return size;
}
```

```
public boolean add(E e) {
   linkLast(e);
   return true;
}
```

LinkedList<E>

```
public E get(int index) {
   checkElementIndex(index);
   return node(index).item;
}

public E set(int index, E element) {
   checkElementIndex(index);
   Node<E> x = node(index);
   E oldVal = x.item;
   x.item = element;
   return oldVal;
}

public void add(int index, E element) {
   checkPositionIndex(index);

   if (index == size)
      linkLast(element);
   else
      linkBefore(element, node(index));
}
```

```
public E remove(int index) {
   checkElementIndex(index);
   return unlink(node(index));
}

private boolean isElementIndex(int index) {
   return index >= 0 && index < size;
}

private void checkElementIndex(int index) {
   if (!isElementIndex(index))
      throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

private boolean isPositionIndex(int index) {
   return index >= 0 && index <= size;
}

private void checkPositionIndex(int index) {
   if (!isPositionIndex(index))
      throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

LinkedList<E>

```
Node<E> node(int index) {
  // assert isElementIndex(index);

  if (index < (size >> 1)) {
    Node<E> x = first;
    for (int i = 0; i < index; i++)
      x = x.next;
    return x;
  } else {
    Node<E> x = last;
    for (int i = size - 1; i > index; i--)
      x = x.prev;
    return x;
  }
}
```

- This is the method that **makes all indexed operations linear** in the number of elements in the list
- Depending whether the index is in the first or the last half of the list, it counts nodes from the start or the end.

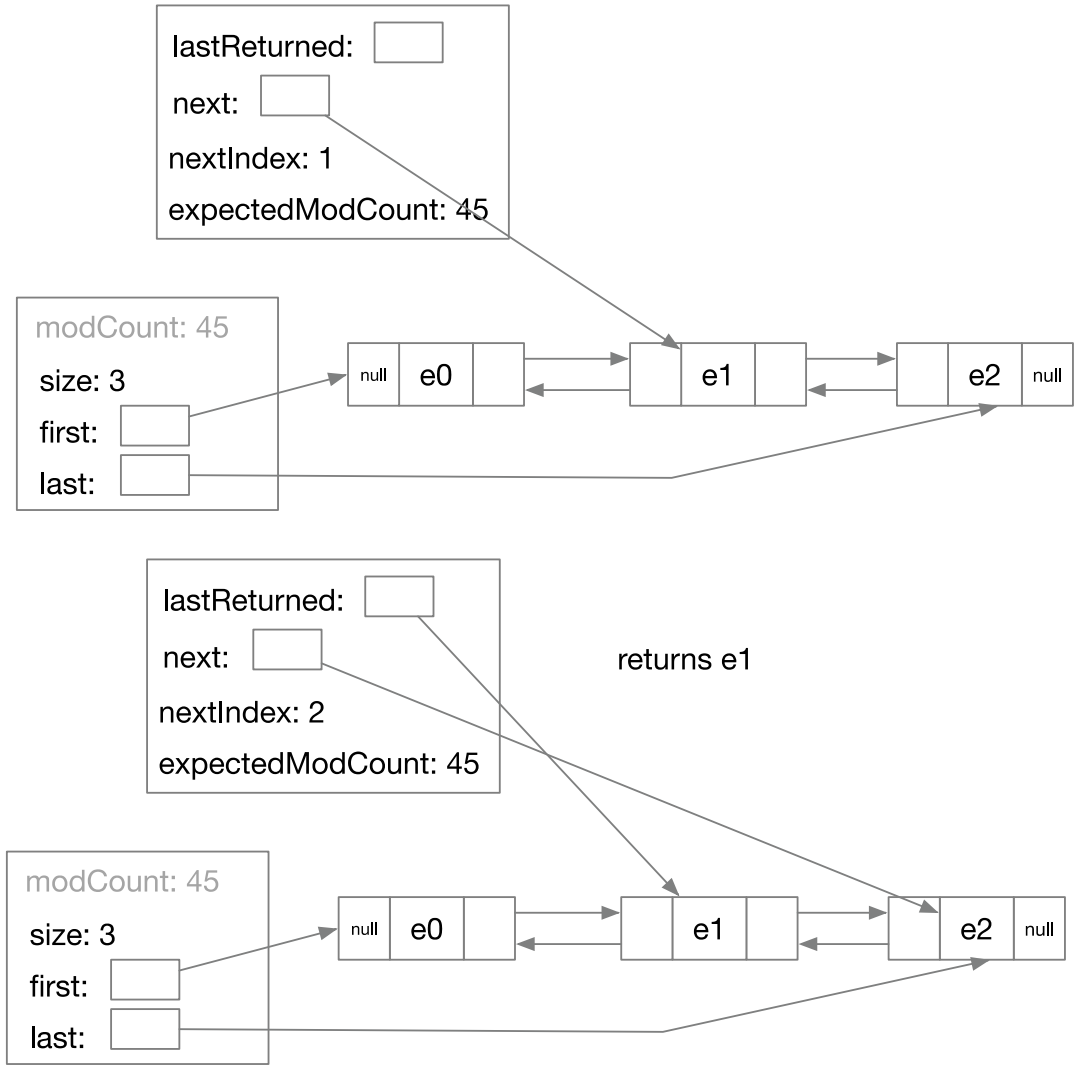
LinkedList<E> (ListIterator<E>)

```
public ListIterator<E> listIterator(int index) {
   checkPositionIndex(index);
   return new ListItr(index);
}

private class ListItr implements ListIterator<E>
{
   private Node<E> lastReturned;
   private Node<E> next;
   private int nextIndex;
   private int expectedModCount = modCount;

   ListItr(int index) {
      // assert isPositionIndex(index);
      next = (index == size) ? null : node(index);
      nextIndex = index;
   }
```

- ListItr is an inner class that implements iterators over LinkedList<E>
  - It's non-static so it has access to the instance of list it is iterating
- It is represented by:
  - **lastReturned**: reference to the node of the last returned element by next / previous (for add / set / remove) or null
  - **next**: node of the element to be returned by next
  - **nextIndex**: index of the element to be returned by next
  - **expectedModCount**: to detect co-modification

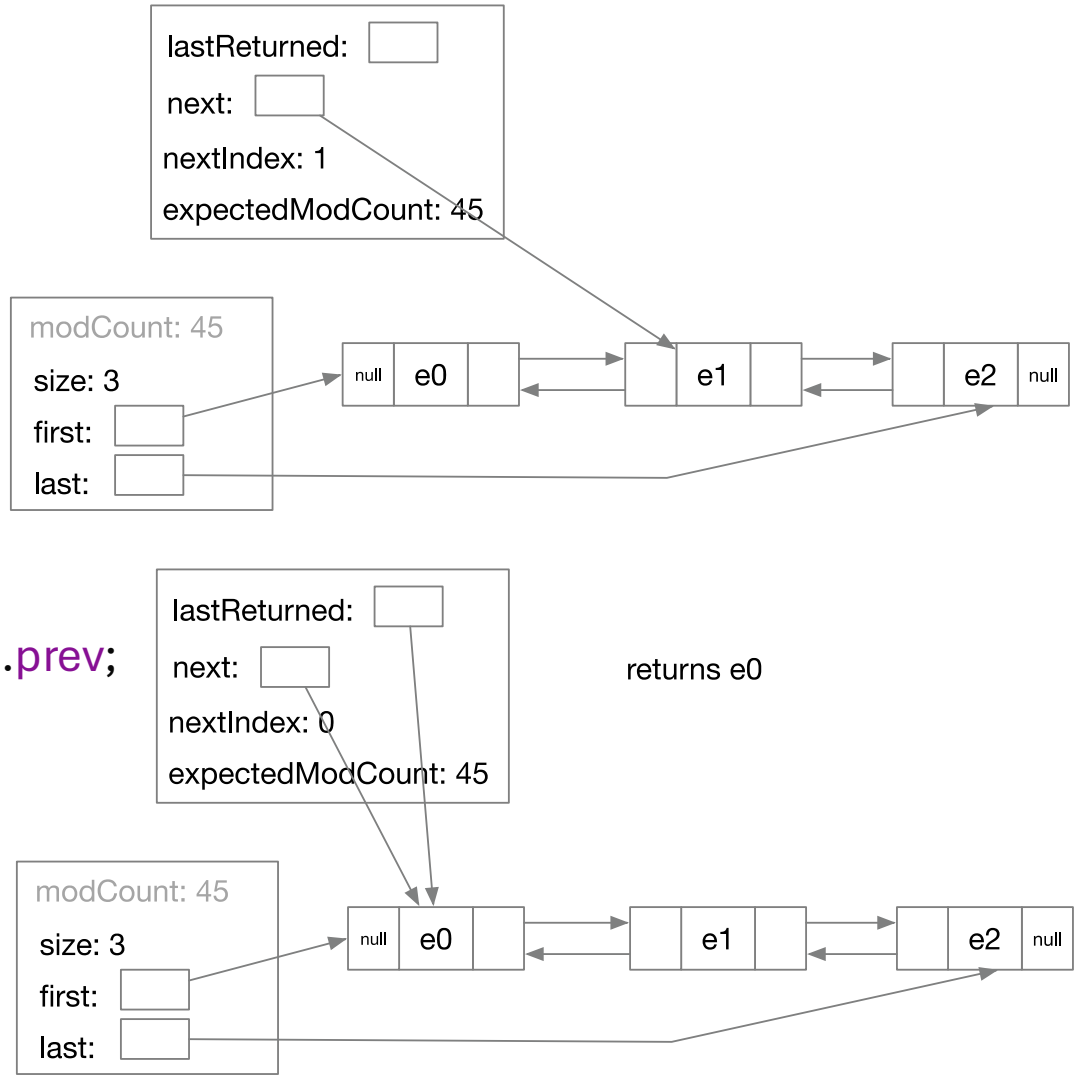
LinkedList<E> (ListIterator<E>)

```
public boolean hasNext() {
   return nextIndex < size;
}

public E next() {
   checkForComodification();
   if (!hasNext())
      throw new NoSuchElementException();

   lastReturned = next;
   next = next.next;
   nextIndex++;
   return lastReturned.item;
}
```



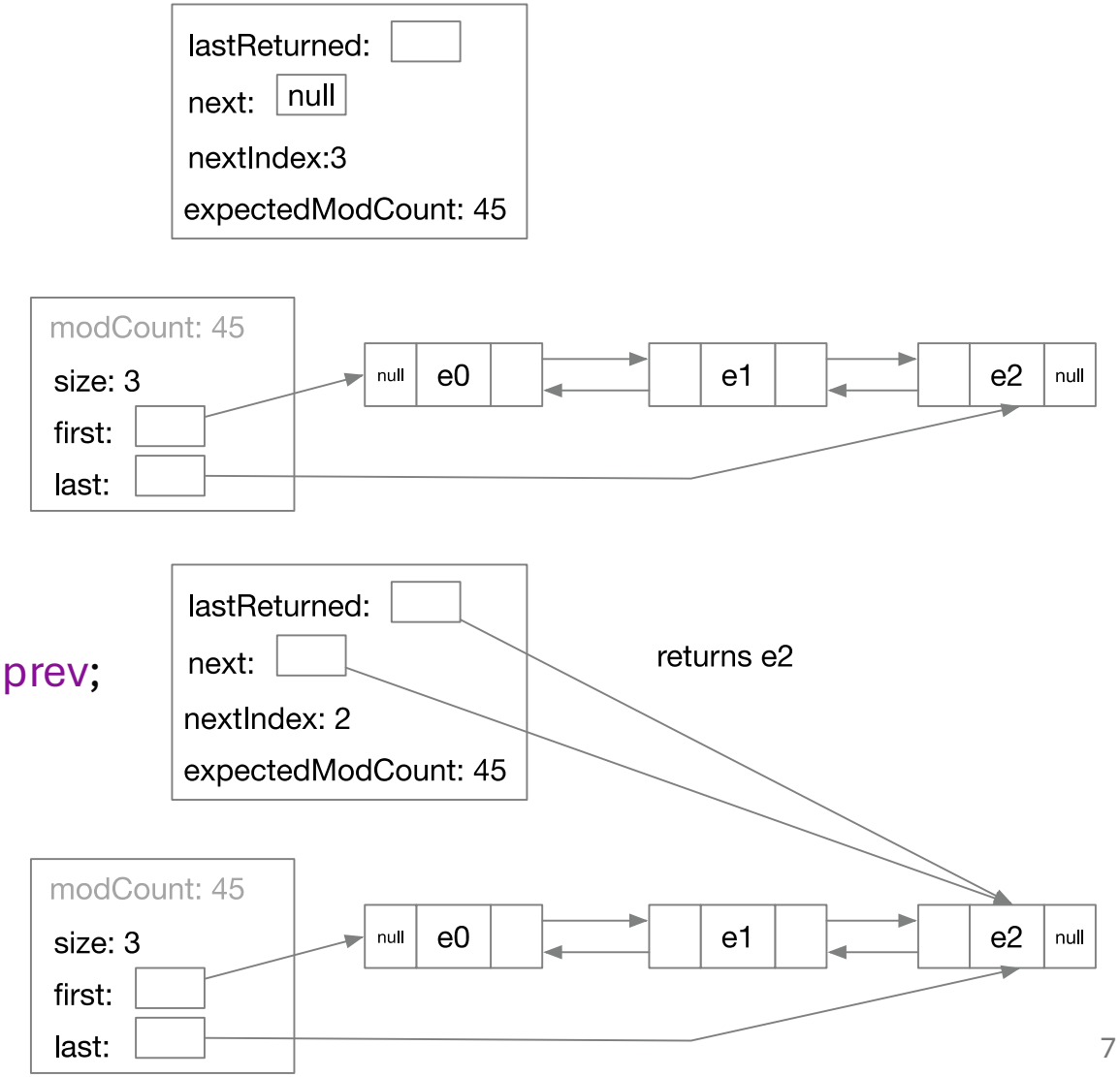
LinkedList<E> (ListIterator<E>)

```
public boolean hasPrevious() {
   return nextIndex > 0;
}

public E previous() {
   checkForComodification();
   if (!hasPrevious())
      throw new NoSuchElementException();

   lastReturned = next = (next == null) ? last : next.prev;
   nextIndex--;
   return lastReturned.item;
}
```



LinkedList<E> (ListIterator<E>)

```
public boolean hasPrevious() {
   return nextIndex > 0;
}

public E previous() {
   checkForComodification();
   if (!hasPrevious())
      throw new NoSuchElementException();

   lastReturned = next = (next == null) ? last : next.prev;
   nextIndex--;
   return lastReturned.item;
}
```



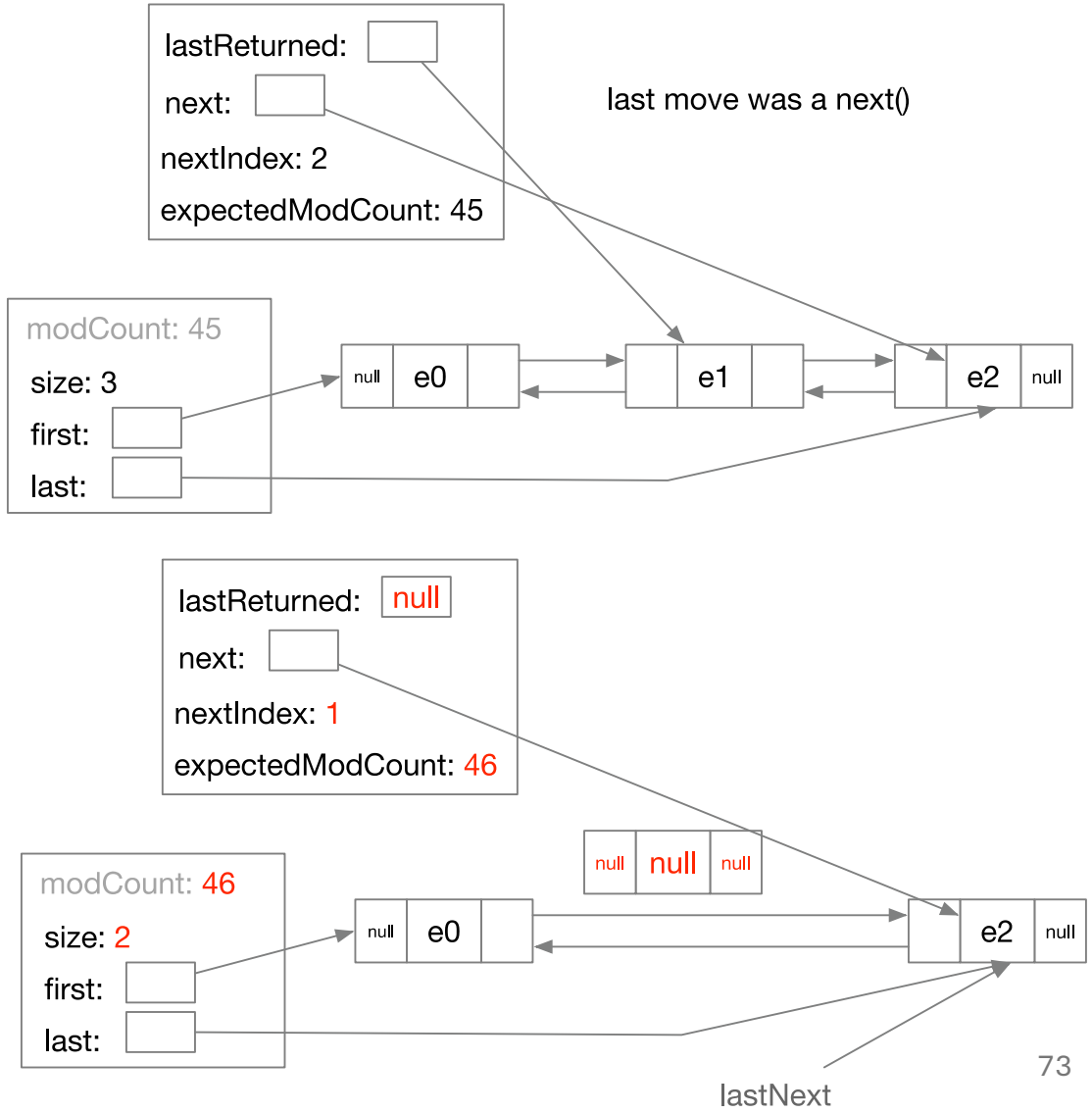
LinkedList<E> (ListIterator<E>)

```
public int nextIndex() {
   return nextIndex;
}

public int previousIndex() {
   return nextIndex - 1;
}

public void set(E e) {
   if (lastReturned == null)
      throw new IllegalStateException();
   checkForComodification();
   lastReturned.item = e;
}
```

```
public void add(E e) {
   checkForComodification();
   lastReturned = null;
   if (next == null)
      linkLast(e);
   else
      linkBefore(e, next);
   nextIndex++;
   expectedModCount++;
}


final void checkForComodification() {
   if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
}
```

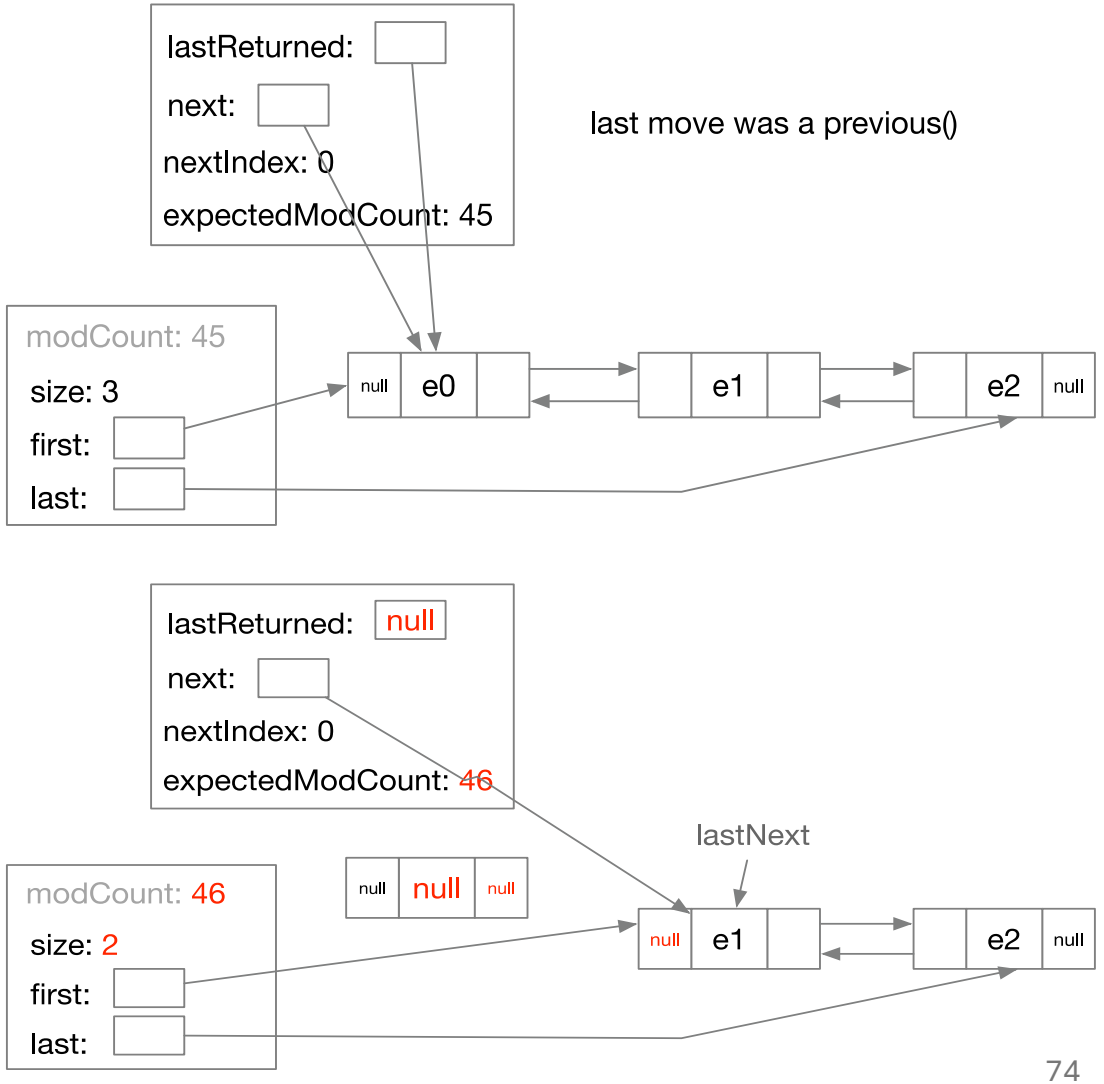
LinkedList<E> (ListIterator<E>)

```
public void remove() {
   checkForComodification();
   if (lastReturned == null)
      throw new IllegalStateException();

   Node<E> lastNext = lastReturned.next;
   unlink(lastReturned);
   if (next == lastReturned)
      next = lastNext;
   else
      nextIndex--;
   lastReturned = null;
   expectedModCount++;
}
```



LinkedList<E> (ListIterator<E>)

```
public void remove() {
   checkForComodification();
   if (lastReturned == null)
      throw new IllegalStateException();

   Node<E> lastNext = lastReturned.next;
   unlink(lastReturned);
   if (next == lastReturned)
      next = lastNext;
   else
      nextIndex--;
   lastReturned = null;
   expectedModCount++;
}
```



Costs of the operations

- The **most important factors** to consider are:
  - **ArrayList<E>** must **move elements** to make room for the new element, or to remove the ”hole” created by a removal
    - And that takes **linear time** $\mathcal{O}(n)$
  - **LinkedList<E>** must do a **sequential search** in all the operations that use the **index**
    - And that takes **linear time** $\mathcal{O}(n)$
  - Besides, in the **ArrayList<E>** implementation, even when we are adding the last element, and we do not need to displace elements, if the array is **full**, we must do a **resize**
    - Another **linear time** $\mathcal{O}(n)$

Amortized Cost

- Let's begin with the **cost of resizing** the array when adding the last element
- This operation is **not always done**, only when the **array is full**
  - **Most of the time it is not needed**, so most of the time, add is $\mathcal{O}(1)$
  - But, sometimes, due to resizing, the operation is $\mathcal{O}(n)$
- **Amortized cost allows** us to **share the cost** of a **sequence of operations** between them
  - The idea is well known in accountability
  - E.g. we compute the cost of a new computer not to the first project we do with it but among all the projects we forecast to do with it during all its live

Amortized Cost

- To simplify the math, we'll consider
  - The initial size of the array is 1
  - We double its size when full
- Suppose we are about to add the element $n + 1$, and the array is full
  - What is the cost of all the copies done until now?
  - $1 + 2 + 4 + 8 + \cdots + n = 2\,n - 1$
- So, this is the cost to amortize among the $n + 1$ insertions
  - $\frac{2n-1}{n+1} = \mathcal{O}(1)$
- This approach to assign amortized costs is called **aggregate analysis** (a most complete strategy is the **potential method**).

Amortized Cost

- Let's generalize this result
  - Initial size is I
  - We'll use $\mathrm{k} > 1$ as the increment factor
- As before, we're about to insert element $n + 1$ and the array is full
  - What is the cost of all the copies done until now?
  - $I \cdot k^0 + I \cdot k^1 + I \cdot k^2 + \cdots + I \cdot n = I \cdot \sum_{i=0}^{log_k\, n} k^i$
  - $I \cdot \sum_{i=0}^{log_k n} k^i = I \cdot \frac{k \cdot k^{log_k \mathrm{n}} - k^0}{k-1} = I \cdot \frac{k \cdot n - 1}{k-1} \approx \mathcal{O}(n)$
- So, as before, the amortized cost is $\frac{\mathcal{O}\,(n)}{n+1} = \mathcal{O}(1)$
  - NOTE: The amortized cost would be linear if, instead of multiplying by k, we'd add a fixed amount of elements at each resize, cause the total cost of the copies would be quadratic.

Costs of the operations on List<E>

Operation	ArrayList<E>	LinkedList<E>
Adding first element	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Adding with index	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Adding last element	$\mathcal{O}(n), \mathcal{O}(1)$ amortized	$\mathcal{O}(1)$
Removing first element	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Removing with index	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Removing last element	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Getting first element	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Getting with index	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Getting last element	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Setting first element	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Setting with index	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Setting last element	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Costs of the operations on List<E>

Operation	ArrayList<E>	LinkedList<E>
Create iterator at first position	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Create iterator at end position	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Create iterator at index	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Adding with iterator	$\mathcal{O}(n)$, $\mathcal{O}(1)$ amortized if at end	$\mathcal{O}(1)$
Removing with iterator	$\mathcal{O}(n)$, $\mathcal{O}(1)$ if at end	$\mathcal{O}(1)$
Getting with iterator	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Setting with iterator	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Costs of the operations on List<E>

- In general, if we do not know the implementation, and we must do sequential operations, use iterators and not indices
  - If you must modify the underlying list, do the operation and the end (no elements are moved even in the ArrayList<E> case)
- Marker interface RandomAccess signals if the implementation provides efficient index access
- If the implementation of the function must use index operation
  - Consider asking for an ArrayList<E> instead of a List<E>

Queue<E> (Deque<E>)

- A Queue<E> is a structure that follows a FIFO policy (First In, First Out)
  - So, you can remove/get the first element, and add the last one
  - A Deque<E> is a structure that combines both allowing adding, removal and access at both ends
- To compare, a Stack<E> follows a LIFO policy (Last In, First Out)
  - We can implement it using both the ideas of ArrayList<E> and LinkedList<E> (using single-linked nodes)
- The LinkedList<E> class implements both Queue<E> and Deque<E> interfaces
- So, can we implement Queue<E> (or Deque<E>) backed by an array data store?

Queue<E>

- It is an extension of Collection<E> which adds operations that follow the FIFO policy
- The two sets of operations differ in the way to signal errors:
  - **add** throws **IllegalStateException** if the queue has capacity restrictions and is full; **offer** returns **false** in this case
  - **remove** / **element** throw **NoSuchElementException** if the queue is empty; **poll** / **peek** return **null** (which can cause ambiguity)

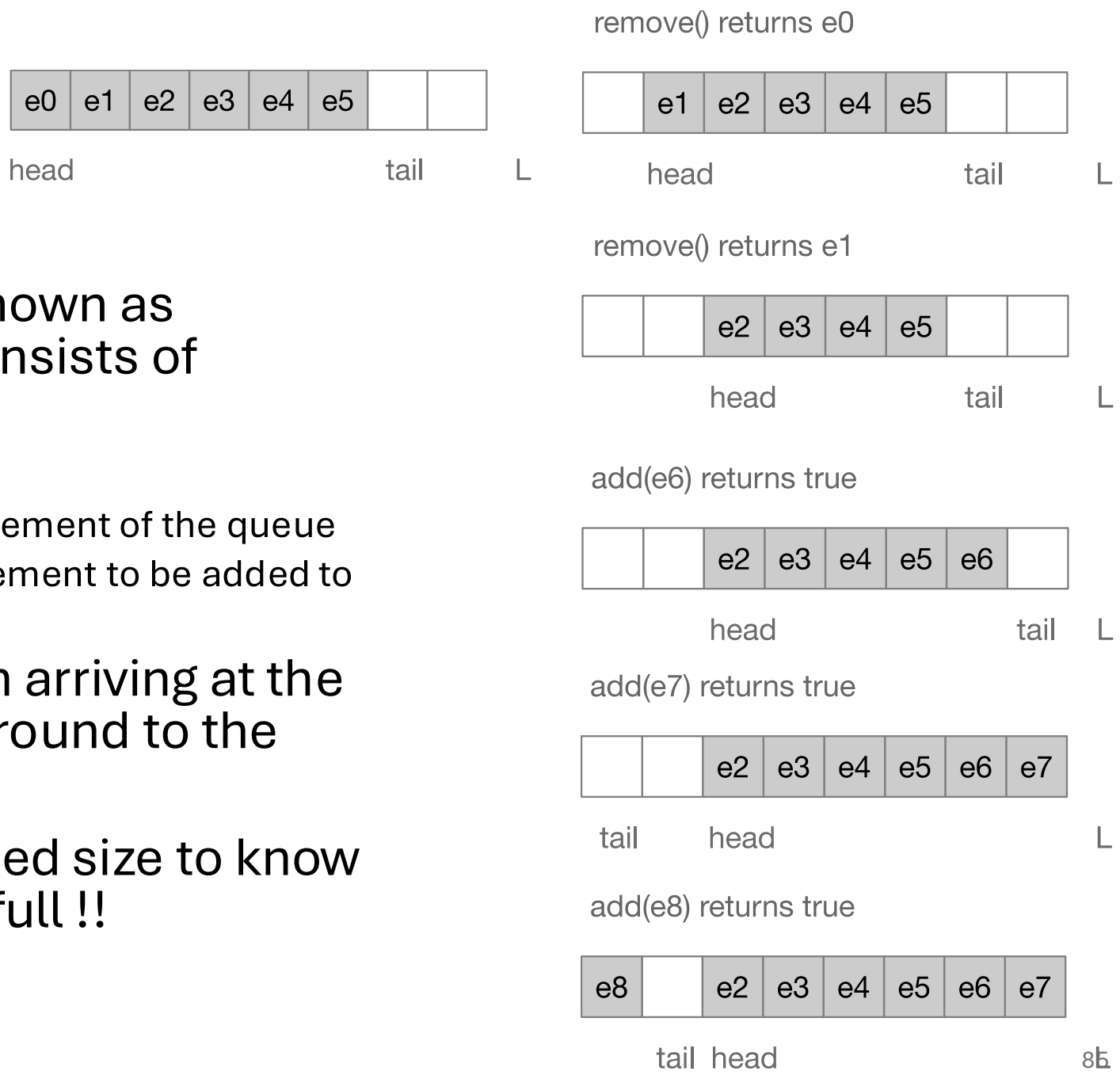
Operation	Throws Exception	Returns special value
Insert (new last element)	boolean add(E e)	boolean offer(E e)
Remove (the first element)	E remove()	E poll()
Examine (the first element)	E element()	E peek()

Queue<E>

- Using directly an ArrayList<E> won’t work.
- If we store the elements in the normal order:
  - add/offer: constant amortized time
  - remove/poll: linear time
  - element/peek: constant time
- If we store the elements in the reverse order:
  - add/offer: linear time
  - remove/poll: constant time
  - element/peek: constant time
- In both cases one of the operations is linear !!

Queue<E>

- We can use a strategy known as **circular array**, which consists of
  - And array of elements
  - Two indexes:
    - head: index of the first element of the queue
    - tail: index for the next element to be added to the queue
- And these indexes, when arriving at the end of the array, come around to the beginning !!
- When head == tail we need size to know if the queue is empty or full !!



ArrayDeque<E>

- This idea is what is used by the class ArrayDeque<E> which implements both Queue<E> and Deque<E>
- The implementation is a little bit much complicated because the array is resizable.
- If you want to understand array manipulation code is a very interesting class to study !!