



# Lesson 9

15.04.2021

```
public class Ex1 {  
    int x = 3;  
  
    public static void main(String[] args) {  
        new Ex1().go1();  
    }  
  
    void go1() {  
        int x;  
        go2(++x);  
    }  
  
    void go2(int y) {  
        int x = ++y;  
        System.out.println(x);  
    }  
}
```

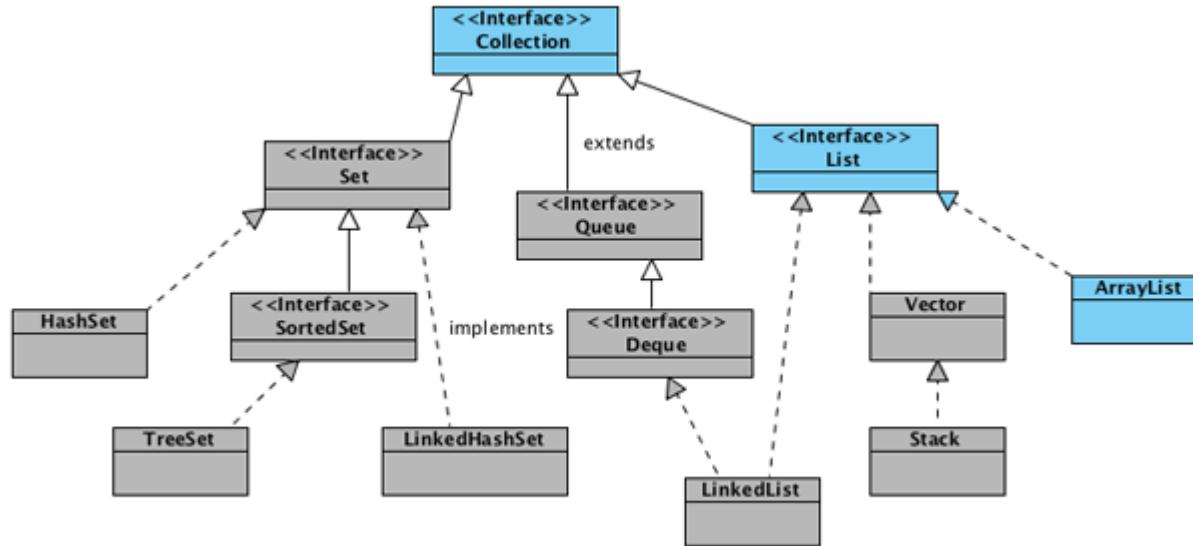
```
public class Ex2 {  
    public static void main(String[] args) {  
        byte[][] bytes = {{1,2,3,4,5}, {1,3,3,4,5,6}};  
        System.out.println(bytes[1].length + " " + bytes.length);  
    }  
}
```

```
public class Ex3 {  
    public static void main(String[] args) {  
        Days d1 = Days.TH;  
        Days d2 = Days.M;  
  
        for (Days d : Days.values()) {  
            if (d.equals(Days.F)) break;  
            d2 = d;  
        }  
        System.out.println((d1 == d2) ? "same" : "not same");  
    }  
    enum Days {M, T, W, TH, F, SA, S}  
}
```

```
public class Ex4 {  
    public static void main(String[] args) {  
        String s = "Hello";  
        String t = new String(s);  
  
        if ("Hello".equals(s)) System.out.println(1);  
        if (t == s) System.out.println(2);  
        if (t.equals(s)) System.out.println(3);  
        if ("Hello" == s) System.out.println(4);  
        if ("Hello" == t) System.out.println(5);  
    }  
}
```

```
public class Ex5 {  
    Ex5() {}  
    Ex5(Ex5 ex) {ex5 = ex;}  
    Ex5 ex5;  
  
    public static void main(String[] args) {  
        Ex5 ex5_1 = new Ex5();  
        Ex5 ex5_2 = new Ex5(ex5_1); ex5_2.go();  
        Ex5 ex5_3 = ex5_2.ex5; ex5_3.go();  
        Ex5 ex5_4 = ex5_1.ex5; ex5_4.go();  
    }  
  
    void go() { System.out.println("hi"); }  
}
```

# ArrayList



***ArrayList*** — реализует интерфейс `List`. Как известно, в Java массивы имеют фиксированную длину, и после того как массив создан, он не может расти или уменьшаться. `ArrayList` может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Элементы `ArrayList` могут быть абсолютно любых типов в том числе и `null`.



## Создание объекта

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

Только что созданный объект list, содержит свойства **elementData** и **size**.

```
elementData = (E[]) new Object[10];
```



Вы можете использовать конструктор **ArrayList(capacity)** и указать свою начальную емкость списка.

## Добавление элементов

```
list.add("0");
```

0	1	2	3	4	5	6	7	8	9
"0"	null								

- 1) проверяется, достаточно ли места в массиве для вставки нового элемента;

```
ensureCapacity(size + 1);
```

- 2) добавляется элемент в конец (согласно значению **size**) массива.

```
elementData[size++] = element;
```

## ensureCapacity(minCapacity)

Если места в массиве не достаточно, новая емкость рассчитывается по формуле  $(oldCapacity * 3) / 2 + 1$ . Второй момент это копирование элементов. Оно осуществляется с помощью **native** метода **System.arraycopy()**, который написан не на Java.

```
// newCapacity - новое значение емкости
elementData = (E[])new Object[newCapacity];
```

```
// oldData - временное хранилище текущего массива с данными
System.arraycopy(oldData, 0, elementData, 0, size);
```

## Добавление в «середину» списка

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
''0''	''1''	''2''	''3''	''4''	''5''	''6''	''7''	''8''	''9''	''10''	''11''	''12''	''13''	''14''	null

```
list.add(5, "100");
```

1) проверяется, достаточно ли места в массиве для вставки нового элемента;

```
ensureCapacity(size+1);
```

2) подготавливается место для нового элемента с помощью **System.arraycopy()**;

```
System.arraycopy(elementData, index, elementData, index + 1, size - index);
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
''0''	''1''	''2''	''3''	''4''	''5''	''5''	''6''	''7''	''8''	''9''	''10''	''11''	''12''	''13''	''14''

3) перезаписывается значение у элемента с указанным индексом.

```
        elementData[index] = element;  
        size++;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
"0"	"1"	"2"	"3"	"4"	"100"	"5"	"6"	"7"	"8"	"9"	"10"	"11"	"12"	"13"	"14"

Как можно догадаться, в случаях, когда происходит вставка элемента по индексу и при этом в вашем массиве нет свободных мест, то вызов **System.arraycopy()** случится дважды: первый в **ensureCapacity()**, второй в самом методе **add(index, value)**, что явно скажется на скорости всей операции добавления.

## Удаление элементов

- по индексу **remove(index)**
- по значению **remove(value)**

```
list.remove(5);
```

1) Сначала определяется какое количество элементов надо скопировать

```
int numMoved = size - index - 1;
```

2) Затем копируем элементы используя **System.arraycopy()**

```
System.arraycopy(elementData, index + 1, elementData, index, numMoved);
```

3) Уменьшаем размер массива и забываем про последний элемент

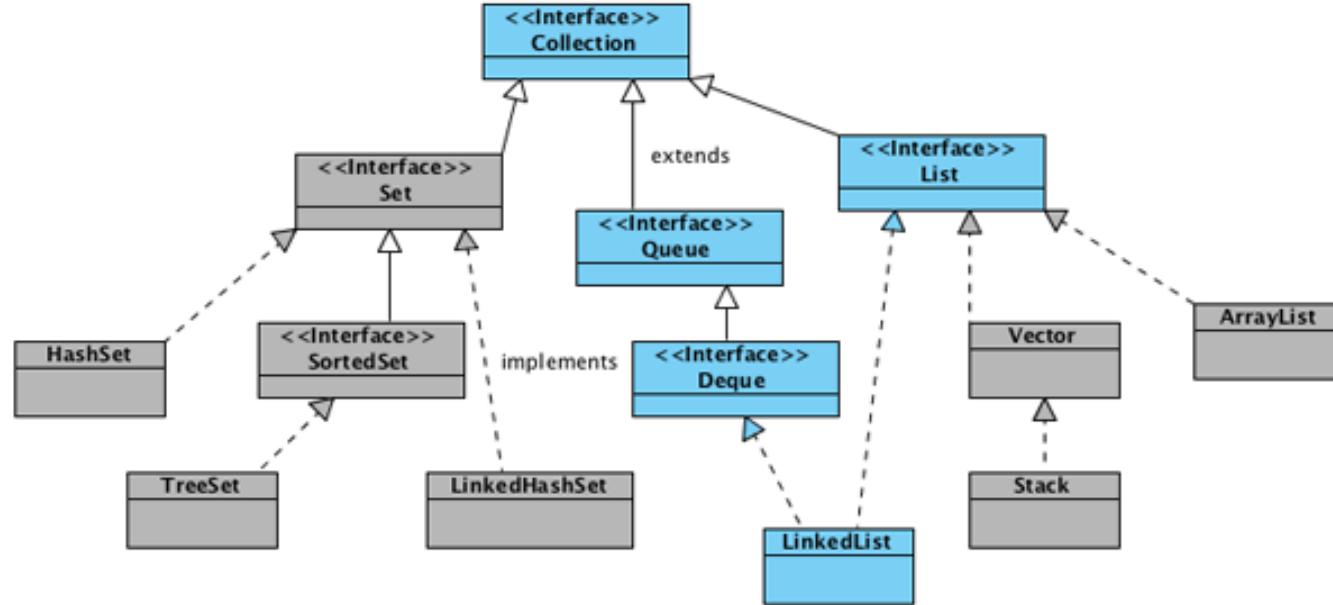
```
elementData[--size] = null; // Let gc do its work
```



При удалении по значению, в цикле просматриваются все элементы списка, до тех пор пока не будет найдено соответствие. Удален будет лишь первый найденный элемент.

- Быстрый доступ к элементам по индексу за время  $O(1)$ ;
- Доступ к элементам по значению за линейное время  $O(n)$ ;
- Медленный, когда вставляются и удаляются элементы из «середины» списка;
- Позволяет хранить любые значения в том числе и null;
- Не синхронизирован.

# LinkedList



***LinkedList*** – реализует интерфейс `List`. Является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы. Итератор поддерживает обход в обе стороны. Реализует методы получения, удаления и вставки в начало, середину и конец списка. Позволяет добавлять любые элементы в том числе и `null`.



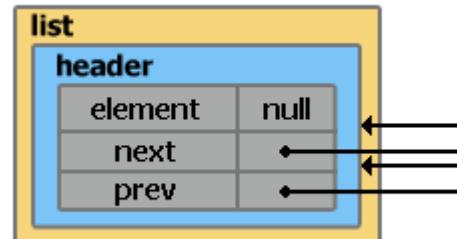
## Создание объекта

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

Только что созданный объект `list`, содержит свойства `header` и `size`.

`header` — псевдо-элемент списка. Его значение всегда равно `null`, а свойства `next` и `prev` всегда указывают на первый и последний элемент списка соответственно. Так как на данный момент список еще пуст, свойства `next` и `prev` указывают сами на себя (т.е. на элемент `header`). Размер списка `size` равен 0.

```
header.next = header.prev = header;
```



## Добавление элементов

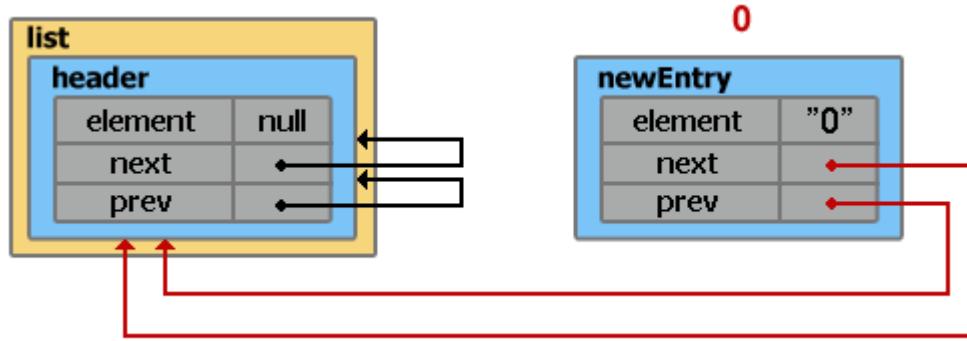
```
list.add("0");
```

Каждый раз при добавлении нового элемента, по сути выполняется два шага:  
1) создается новый новый экземпляр класса **Entry**

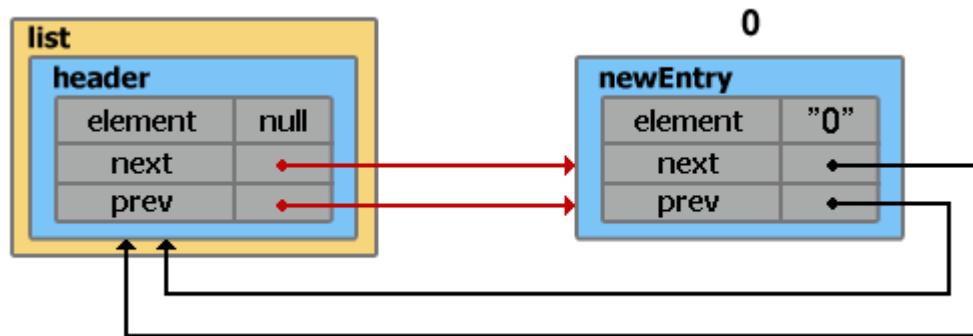
```
private static class Entry<E>
{
    E element;
    Entry<E> next;
    Entry<E> prev;

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

```
Entry newEntry = new Entry("0", header, header.prev);
```



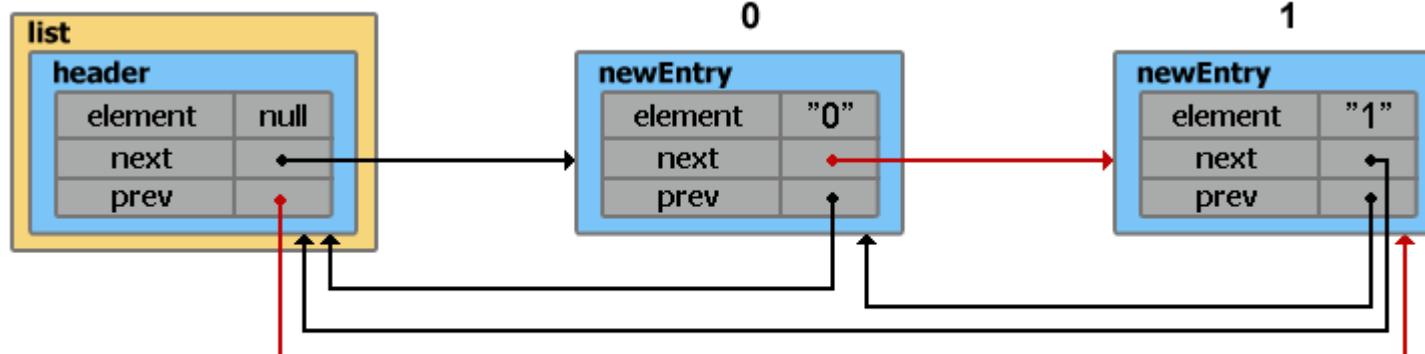
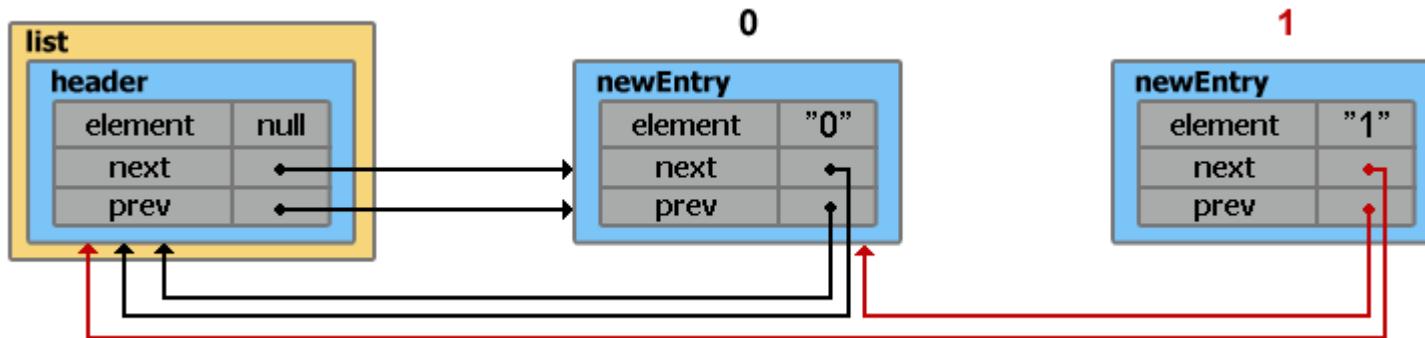
2) переопределяются указатели на предыдущий и следующий элемент



list.add("1");

// header.prev указывает на элемент с индексом 0

```
Entry newEntry = new Entry("1", header, header.prev);
```

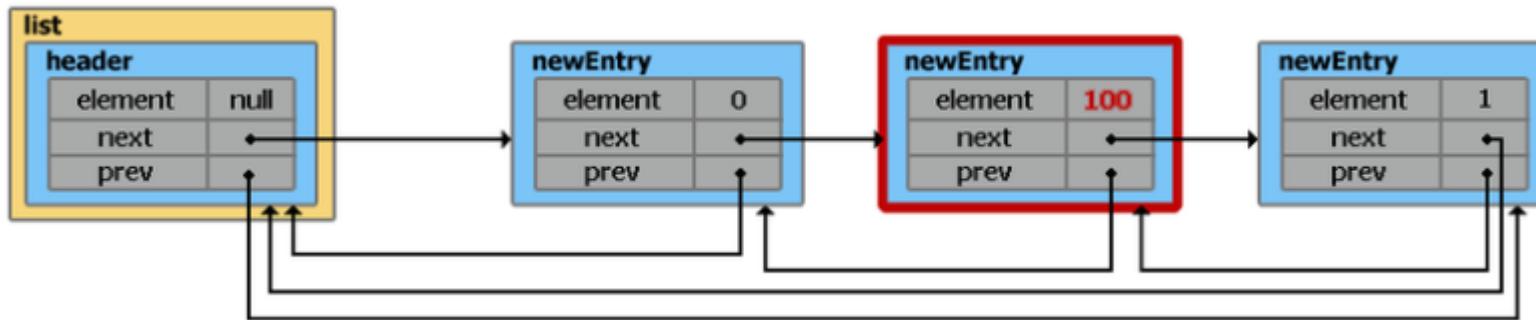


## Удаление элементов

- из начала или конца списка с помощью `removeFirst()`, `removeLast()` за время  $O(1)$ ;
- по индексу `remove(index)` и по значению `remove(value)` за время  $O(n)$ .

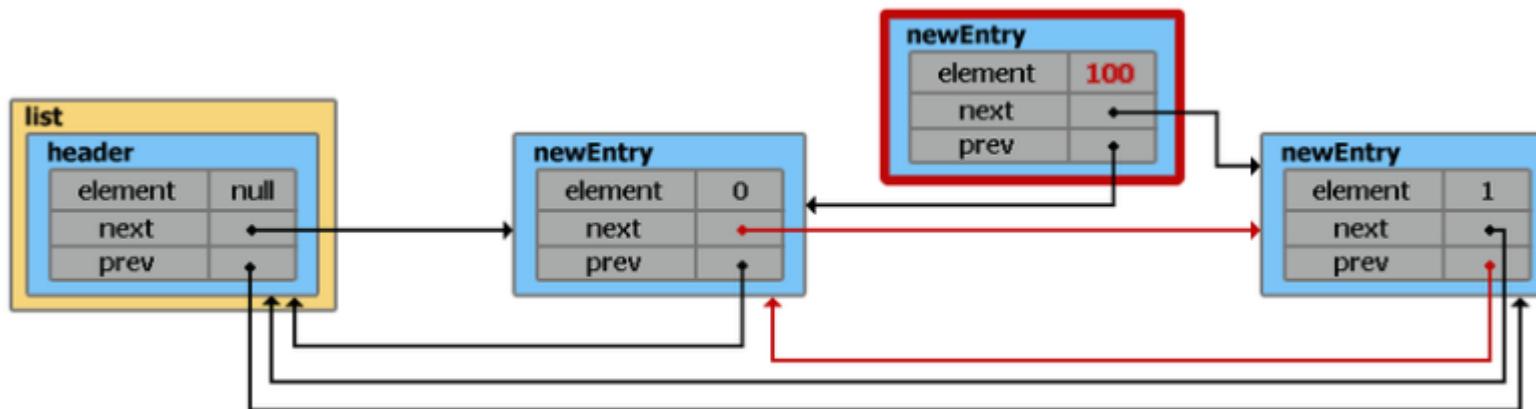
```
list.remove("100");
```

- 1) поиск первого элемента с соответствующим значением



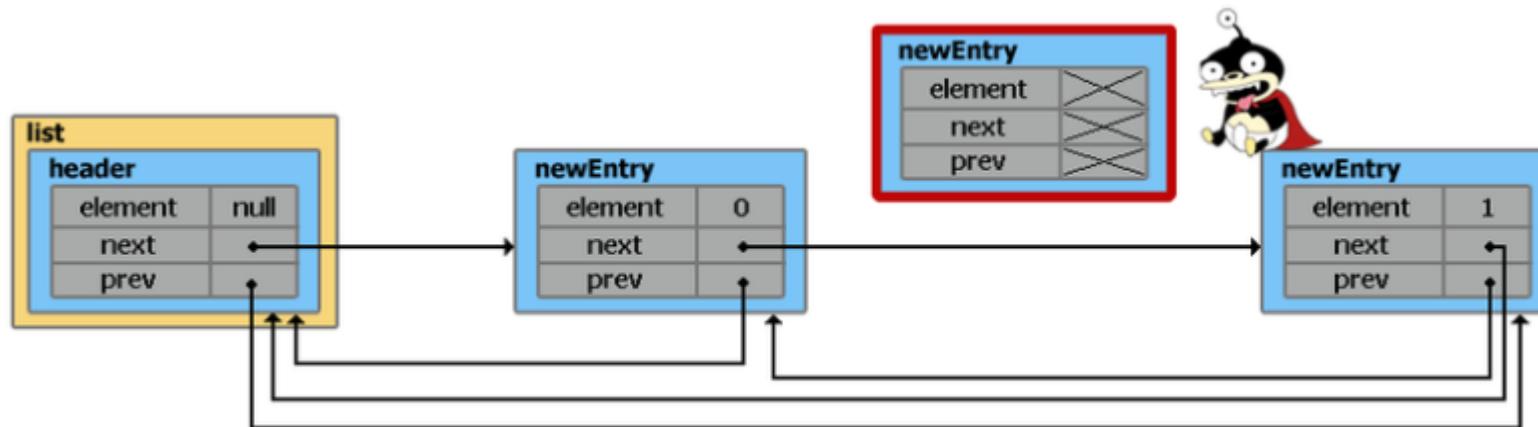
## 2) переопределяются указатели на предыдущий и следующий элемент

```
// Значение удаляемого элемента сохраняется
// для того чтобы в конце метода вернуть его
E result = e.element;
e.prev.next = e.next;
e.next.prev = e.prev;
```



### 3) удаление указателей на другие элементы и предание забвению самого элемента

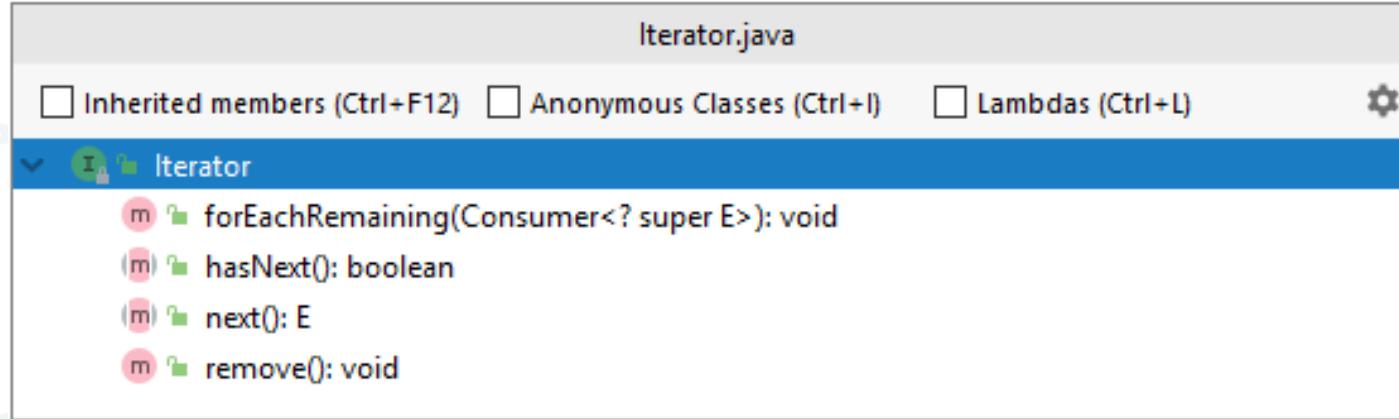
```
e.next = e.prev = null;  
e.element = null;  
size--;
```



- Из `LinkedList` можно организовать стэк, очередь, или двойную очередь, со временем доступа  $O(1)$ ;
- На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время  $O(n)$ . Однако, на добавление и удаление из середины списка, используя `ListIterator.add()` и `ListIterator.remove()`, потребуется  $O(1)$ ;
- Позволяет добавлять любые значения в том числе и `null`. Для хранения примитивных типов использует соответствующие классы-оберки;
- Не синхронизирован.

## Iterator

Итератор способен поочередно обойти все элементы в коллекции. При этом он позволяет это сделать без вникания во внутреннюю структуру и устройство коллекций.



Методы, которые должен имплементировать `Iterator`:

**boolean hasNext()** — если в итерируемом объекте (пока что это `Collection`) остались еще значение — метод вернет `true`, если значения кончились `false`.

**E next()** — возвращает следующий элемент коллекции (объекта). Если элементов больше нет (не было проверки `hasNext()`, а мы вызвали `next()`, достигнув конца коллекции), метод бросит `NoSuchElementException`.

**void remove()** — удалит элемент, который был в последний раз получен методом `next()`. Метод может бросить:

*UnsupportedOperationException*, если данный итератор не поддерживает метод `remove()` (в случае с `read-only` коллекциями, например)

*IllegalStateException*, если метод `next()` еще не был вызван, или если `remove()` уже был вызван после последнего вызова `next()`.