MACQUARIE
University

WCOM125/COMP125

Fundamentals of Computer Science

Lists

Gaurav Gupta

July 11, 2018

Contents

List of exercises

Section 1.

Why, oh why?

Before we do anything hasty, let's take a look at why what we have so far (arrays) is just not good enough. Mainly, two reasons,

1. **Inability to re-size**: We cannot re-size an array as and when required. An example:

   EXAMPLE: Storing all negative items from an array into another array

   Again, there is no way of knowing how many negative values are there in the source array. So,

   a) go through the source array once to count how many records are there,

   b) create an array of that size,

   c) go through the source array again and copy the negative items into the new array

2. **Lack of operations/methods**: There are only two operations we can perform on an array **arr** are:

   a) **arr.length** : gives number of items in the array

   b) **arr[i]** : returns item at index **i** (throws **ArrayIndexOutOfBoundsException** if index invalid).

   But there are so many other operations we use very frequently. Like:

   a) Checking if the array contains an item

   b) Getting index of the first occurrence (or last occurrence) of an item

   c) Adding/removing an item to/from anywhere in the array

   d) Getting a sub-array (from one index to another)

   Not only are these operations required on their own, they are beneficial to write more complex methods. For example, determining if an item exists exactly once can be reduced to checking if index of first occurrence of the item equals index of last occurrence of the item (and neither is -1).

Section 2.

The List Interface (adapted from official Java Tutorials)

A List is an ordered **Collection** . Lists may contain duplicate elements. In addition to the operations inherited from **Collection** (we'll just mention these instead of going into the details), the List interface includes operations for the following:

1. **Positional access** : manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
2. **Search** : searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
3. **Iteration** : extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behaviour.
4. **Range-view** : The sublist method performs arbitrary range operations on the list.

The Java platform contains two general-purpose List implementations.

- **ArrayList** (Resizable-array implementation of the **List** interface), which is usually the better-performing implementation, and,
- **LinkedList** (Doubly-linked list implementation of the **List** and **Deque** interfaces) which offers better performance under certain circumstances.

2.1 Creating List objects

A List is collection of objects, and not variables of primitive data types. We can create lists containing **Rectangle** objects, **Time** objects, **Matrix** objects, but NOT **int** variables, **char** variables, **double** variables, or **boolean** variables.

So how can one create a collection of, say, integers?

Java provides a *wrapper* class for each primitive data type. For data type int, the wrapper class is **Integer** .

Following is an example of using the **Integer** class:

```
1  int x = 5;
2  Integer y = 5; //just like that!
3  Integer z = x; //can copy int into Integer
4  Integer result = y + z; //all primitive operators can be
5                          //used with Integer objects
6  int a = result; //Integer objects can be copied back into int
```

Assuming that **T** is a class (This is a very standard notation throughout the official java documentation), the syntax to create a List object is:

```
1 import java.util.ArrayList;
2 ...
3 ArrayList<T> list = new ArrayList<T>(); //prior to JDK 7
4 ArrayList<T> list = new ArrayList(); //JDK 7 and later
```

Similarly for **LinkedList** ,

```
1 import java.util.LinkedList;
2 ...
3 LinkedList<T> list = new LinkedList<T>(); //prior to JDK 7
4 LinkedList<T> list = new LinkedList(); //JDK 7 and later
```

2.1.1 Example of creating List objects

```
1  ArrayList<Integer> myList = new ArrayList();
2  LinkedList<String> yourList = new LinkedList();
```

2.2 Important methods applicable to both kinds of list

Following are some important[1] methods can be called on objects of both kinds of lists (assuming **list** is an object of either **ArrayList** or **LinkedList**):

1. **list.size()** : returns number of items in the list.

EXAMPLE

```
1  //assuming list of String objects = ["this", "is", "so", "cool"]
2  int s = list.size(); //s = 4
```

[1] As in, we'll be using these very often

2. **list.get(int index)** : returns item at passed index [2].

EXAMPLE

```
//assuming list of Boolean objects = [true, false, true, false, false]
boolean a = list.get(0); //a = true
boolean b = list.get(3); //b = false
boolean c = list.get(6); //throws IndexOutOfBoundsException
```

3. **list.add(<T> obj)** : adds the passed object at the end of the list.

EXAMPLE

```
//assuming list of Double objects = [1.5, 4.5, 2.5]
list.add(3.5); //now list = [1.5, 4.5, 2.5, 3.5]
```

4. **list.add(int index, <T> obj)** : adds the passed object at the passed index[2].

EXAMPLE

```
//assuming list of Double objects = [1.5, 4.5, 2.5]
list.add(2, 3.5); //now list = [1.5, 4.5, 3.5, 2.5]
list.add(0, 7.5); //now list = [7.5, 1.5, 4.5, 3.5, 2.5]
list.add(-4, 3.5); //throws IndexOutOfBoundsException
```

5. **list.remove(int index)** : removes and returns item at passed index[2].

EXAMPLE

```
//assuming list of Integer objects = [10, 20, 30, 50, 60]
list.remove(0); //now list = [20, 30, 50, 60]
list.remove(list.size()-1); //now list = [20, 30, 50]
```

6. **list.remove(<T> obj)** : removes the object if it exists and returns **true** . Returns **false** if object doesn't exist in the list.

EXAMPLE

```
//assuming list of Integer objects = [10, 20, 30, 50, 60]
list.remove(30); //that's to remove item at INDEX 30
//therefore, it throws IndexOutOfBoundsException
Integer itemToRemove = 30;
list.remove(itemToRemove); //now list = [10, 20, 50, 60]
```

[2]Throws **IndexOutOfBoundsException** if index is not valid

7. **set(int index, <T> obj)** : Replaces the item at passed index by passed object[2].

EXAMPLE

```
//assuming list of Character objects = ['l', 'g', 'b', 't', 'q', 'i', 'a']
list.set(5, 'p'); //now list = ['l', 'g', 'b', 't', 'q', 'p', 'a']
list.set(8, '+'); //throws IndexOutOfBoundsException
```

2.2.1 Honorary mention

Some other methods which are quite useful are:

1. **list.contains(<T> obj)** : returns **true** if object exists in list, **false** if not.
2. **list.indexOf(<T> obj)** : returns index of **first** occurrence of object in list, -1 if object doesn't exist in list
3. **list.lastIndexOf(<T> obj)** : returns index of **last** occurrence of object in list, -1 if object doesn't exist in list
4. **list.subList(int startIndex, int endIndex)** : returns a sub-list from **startIndex** (inclusive) to **endIndex** (exclusive).

2.2.2 Additional resources for List objects

1. The method **toString** is meaningfully defined in the **List** class, so you can display a list or store it's state in a String as,

EXAMPLE

```
//assuming list of Integer objects = [10, 70, 20, 90]
System.out.println(list); //displays [10, 70, 20, 90]
String state = list.toString(); //state = "[10, 70, 20, 90]"
```

2. **for-each** loop. Traversing a list using counter is fine, but unnecessarily complex. Instead, Java provides a loop specifically for Collections.

   It's like saying

   "for each item (current) in the collection, access current"

   The **for-each** loop is useful when you don't need the index for any purpose besides accessing an item. The syntax of this loop is:

```
        //assuming list is a List object
        for(<T> item: list) {
                //item holds reference to current item
        }
```

Following are two examples, the first where the **for-each** loop is useful, and the second where it cannot be used without making life more complex (as we NEED the index)

EXAMPLE 1: Adding items in list (for-each useful)

```
//assuming list of Integer objects = [10, 70, 20, 90]
int total = 0;
for(Integer item: list) {
      total = total + item;
}
//total = 10+70+20+90 = 190
```

EXAMPLE 2: Finding index of first negative item (for-each not so useful)

```
//assuming list of Integer objects = [10, 70, -20, -90]
int idx = -1;
for(int i=0; i < list.size() && idx==-1; i++) {
      if(list.get(i) < 0) {
              idx = i;
              //as soon as idx changes, idx==-1 becomes false
      }
}
//idx = 2
```

2.3 A more comprehensive example

The purpose of this program is to,

- load 1000 random values within a range into a list
- find the average

- find the index of the highest value
- remove any duplicates

PART 1 - Loading values

```
import java.util.*; //both ArrayList and Random are in util package
...
int min = 25;
int max = 10000;
Random randomizer = new Random(); //random number generator
ArrayList<Integer> list = new ArrayList<>();
for(int i=0; i < 1000; i++) {
        int randomNumber = min + randomizer.nextInt(max - min + 1);
        list.add(randomNumber);
}
```

PART 2 - Finding the averages

```
int total = 0;
for(Integer item: list) {
        total = total + item;
}
double average = total / list.size();
```

PART 3 - Finding index of highest value

```
int result = 0;
for(int i=0; i < list.size(); i++) { //need index so counter-based
    loop
        if(list.get(i) > list.get(maxIndex)) {
                maxIndex = i;
        }
}
System.out.println("Highest value at index "+maxIndex);
```

PART 4 - Removing duplicates (Version 1)

```
for(int i=0; i < list.size(); i++) {
        for(int k=i+1; k < list.size(); k++) {
                if(list.get(i) == list.get(k) {
                        list.remove(k); //remove item at index k
                        k--; //to check next item moved into k again
                }
        }
}
```

PART 4 - Removing duplicates (Version 2)

```
for(int i=0; i < list.size(); i++) {
  int c = list.get(i);
  if(list.indexOf(c) != list.lastIndexOf(c)) { //multiple occurrences
        list.remove(i); //remove item at index i
        i--; //to check next item moved into i again
  }
}
```

PART 4 - Removing duplicates
(Version 3 - space heavy but who cares :D)

```
ArrayList<Integer> temp = new ArrayList<>();
for(Integer item: list) {
        if(!temp.contains(item)) {
                temp.add(item);
        }
}
list = temp; //update reference of your list
```

Section 3.

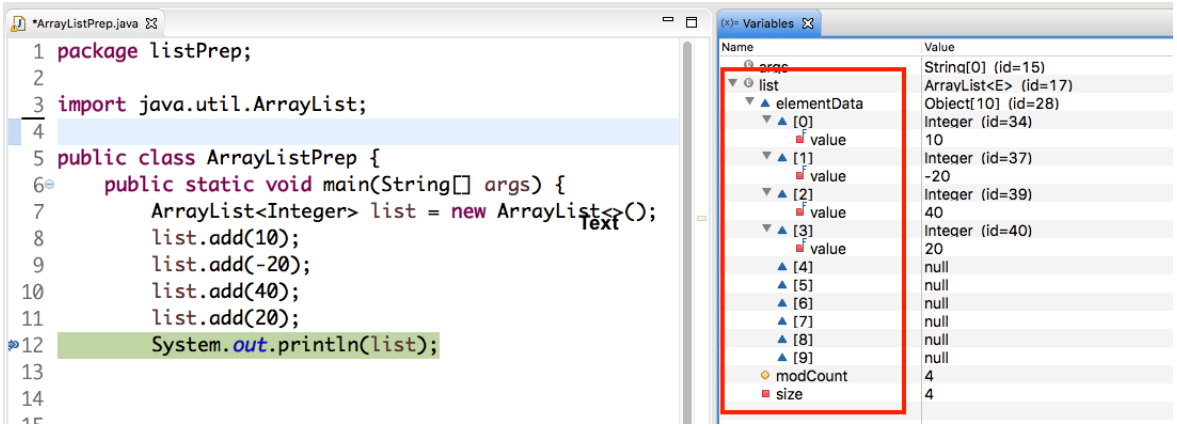
Difference between ArrayList and LinkedList

While there are some minor differences in the behaviours of the two, the important difference is in terms of how the two store items in the collection.

- **ArrayList** has an instance variable array, **elementData** (illustrated in the diagram below), in which the items are stored. This array is re-sized as and when required.

- **LinkedList** on the other hand stores the items as *linked nodes*. We will talk about these a bit later.

Figure 1: Storage of items in an ArrayList object



Section 4.

Custom Implementation of the essence of ArrayList class

4.1 Instance variables required

The essential component of ArrayList class is that it should hold the items in an array, that *grows* as and when required.

The maximum number of items that can be stored in the array is given by **data.length** .

But we also need to store how many items have been added so far.

So our instance variables are:

```
class MyArrayList {
        private int[] data;
        private int nItems;
}
```

4.2 Adding items to the list

Next, we need to have a method that adds an item to the list. For this, the core operation is,

```
data[nItems] = itemToAdd;
nItems++;
```

But what if the array is full? That is, items added so far equals the size of the array. Better to have that in a method of its own.

4.2.1 Checking if array is full

```
public boolean isFull() {
        if(nItems == data.length)
                return true;
        else
                return false;
}
```

If the array is full, we need to grow it. Growing an array is a 3-step process.

4.2.2 Growing the array

Step 1: Create a bigger temporary array

```
int[] temp = new int[data.length + 10];
```

Step 2: Copy over all items from your array to temporary array

```
for(int i=0; i < data.length; i++) {
        temp[i] = data[i];
}
```

Step 3: Copy reference held by temporary array into your array

```
data = temp;
```

Putting this into one method,

```
public void grow() {
        int[] temp = new int[data.length + 10]; //create bigger array

        for(int i=0; i < data.length; i++) { //copy items over
                temp[i] = data[i];
        }

        data = temp; //re-reference instance variable array
}
```

In our **add** method, we can check if array is full, and if it is, we grow it and proceed as normal.

```
public void add(int item) {
        if(isFull())
                grow();
        //now not full anymore :)
        data[nItems] = item;
        nItems++;
}
```

4.3 Removing an item

While removing an item, you have a choice, to re-size the array to a smaller array or not. This is purely subjective and depends on what's more important - Time or Space? If time, then don't waste time in shrinking the array), and if space, then shrink the array. In our example, we don't (shrink the array).

We also have to move all subsequent items one space towards the beginning of the array.

```
public Integer remove(int index) {
        if(index < 0 || index >= nItems) {
                return null; //advantage of return type being class
        }
        else {
                int result = data[index];
                for(int i=index; i < nItems - 1; i++) {
                        data[i] = data[i+1];
                }
                nItems--;
                return result;
        }
}
```

As the last step, we will add two more methods.

1. One that adds an item at a specific index. Just like remove, all subsequent items need to be moved towards the back (end) of the array by one index.

```
public boolean add(int index, int item) {
        if(index < 0 || index > nItems)
                return false; //failure
        if(isFull())
                grow();
        for(int i=nItems; i > index; i--) {
                data[i] = data[i-1];
        }
        data[index] = item;
        return true;
}
```

2. Good old toString!

```
public String toString() {
        if(nItems == 0)
                return "[]";
        String result = "[";
        for(int i=0; i < nItems; i++) {
                result = result + data[i] + ",␣";
        }
        int n = result.length();
        result = result.substring(0, n-2); //remove the last ", "
        result = result + "]";
        return result;
}
```

Putting it all together (and compacting a little bit),

```
class MyArrayList {
        private int[] data;
        private int nItems;

        public Integer get(int index) {
                if(index < 0 || index >= nItems)
                        return null;
                return data[index];
        }

        public boolean isFull() {
                return (nItems == data.length);
                //returns outcome of expression
        }


        //continued on next page

```

```
19
20          public void grow() {
21                  int[] temp = new int[data.length + 10];
22                  for(int i=0; i < data.length; i++) {
23                          temp[i] = data[i];
24                  }
25                  data = temp;
26          }
27
28          public void add(int item) {
29                  if(isFull())
30                          grow();
31                  data[nItems] = item;
32                  nItems++;
33          }
34
35          public Integer remove(int index) {
36                  if(index < 0 || index >= nItems)
37                          return null;
38                  int result = data[index];
39                  for(int i=index; i < nItems - 1; i++)
40                          data[i] = data[i+1];
41                  nItems--;
42                  return result;
43          }
44  }
```

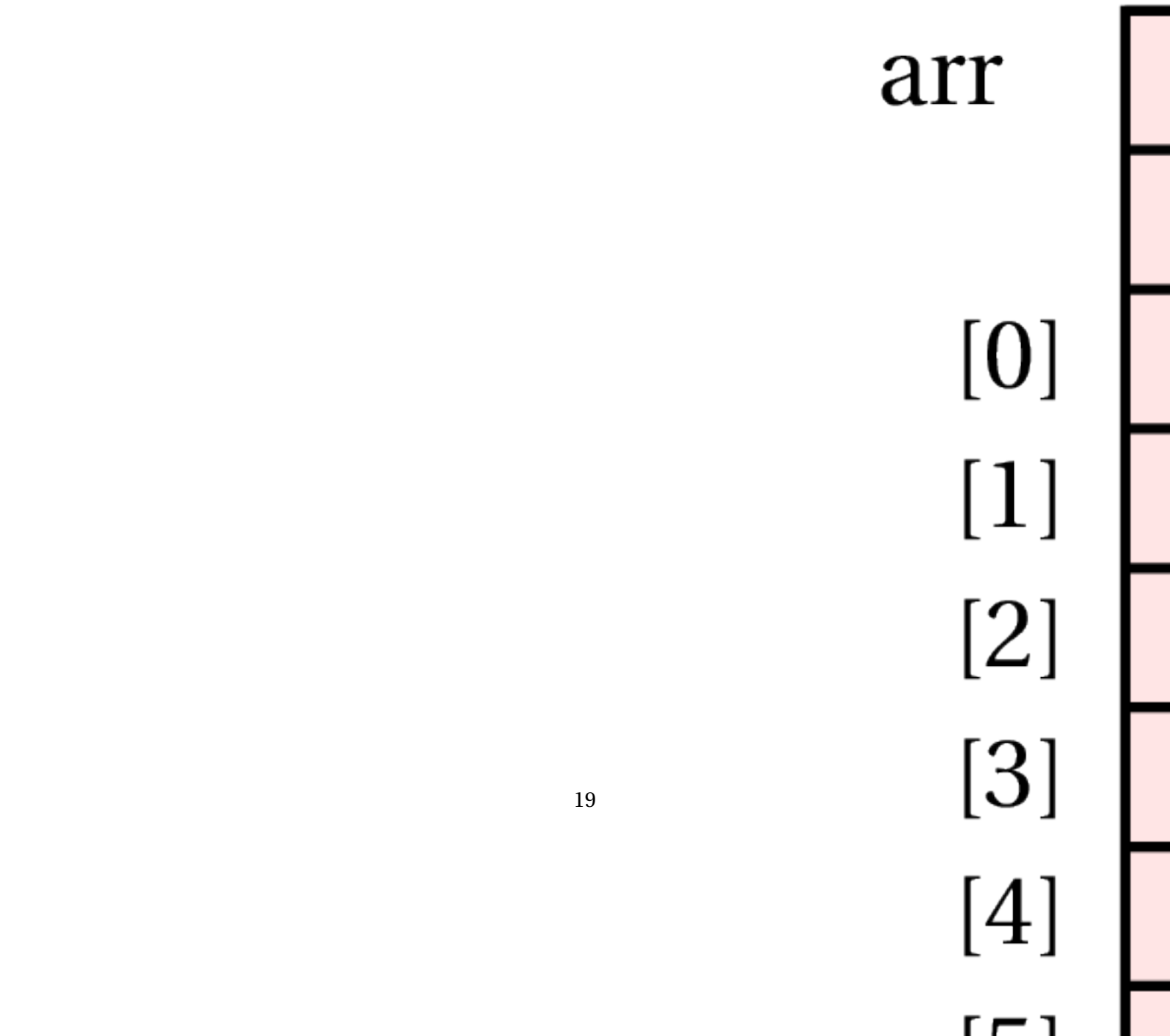
4.4 Time complexity analysis

4.4.1 Accessing an item

We need to understand how an array is stored to understand how array items are accessed.

An array is stored as a contiguous block of memory.

As an example, if there is an array of 6 integers, 24 bytes are reserved (4 per integer). The first item is stored in the first four bytes, the second item in the next four bytes and so on.


arr
[0]
[1]
[2]
[3]
[4]
[5]

Thus,

- The address of the first item (at index 0) is the base address of the array (in this example, 600).
- The address of the second item (at index 1) is the base address of the array + 4 bytes.
- The address of the third item (at index 2) is the base address of the array + 2 * 4 bytes.
- ...
- The address of the last item (at index 5) is the base address of the array + 5 * 4 bytes.

With a little tweak we can see:

- Address(arr[**0**]) = baseAddress + **0** * 4
- Address(arr[**1**]) = baseAddress + **1** * 4
- Address(arr[**2**]) = baseAddress + **2** * 4
- ...
- Address(arr[**arr.length - 1**]) = baseAddress + (**arr.length - 1**) * 4

Putting it all together,

```
(address of item at index idx) = (base address of array) +
                                 (size of each item) * (idx);
```

4.4.2 Example: Accessing an item

Let's say we want to access **arr[3]** in the following statement:

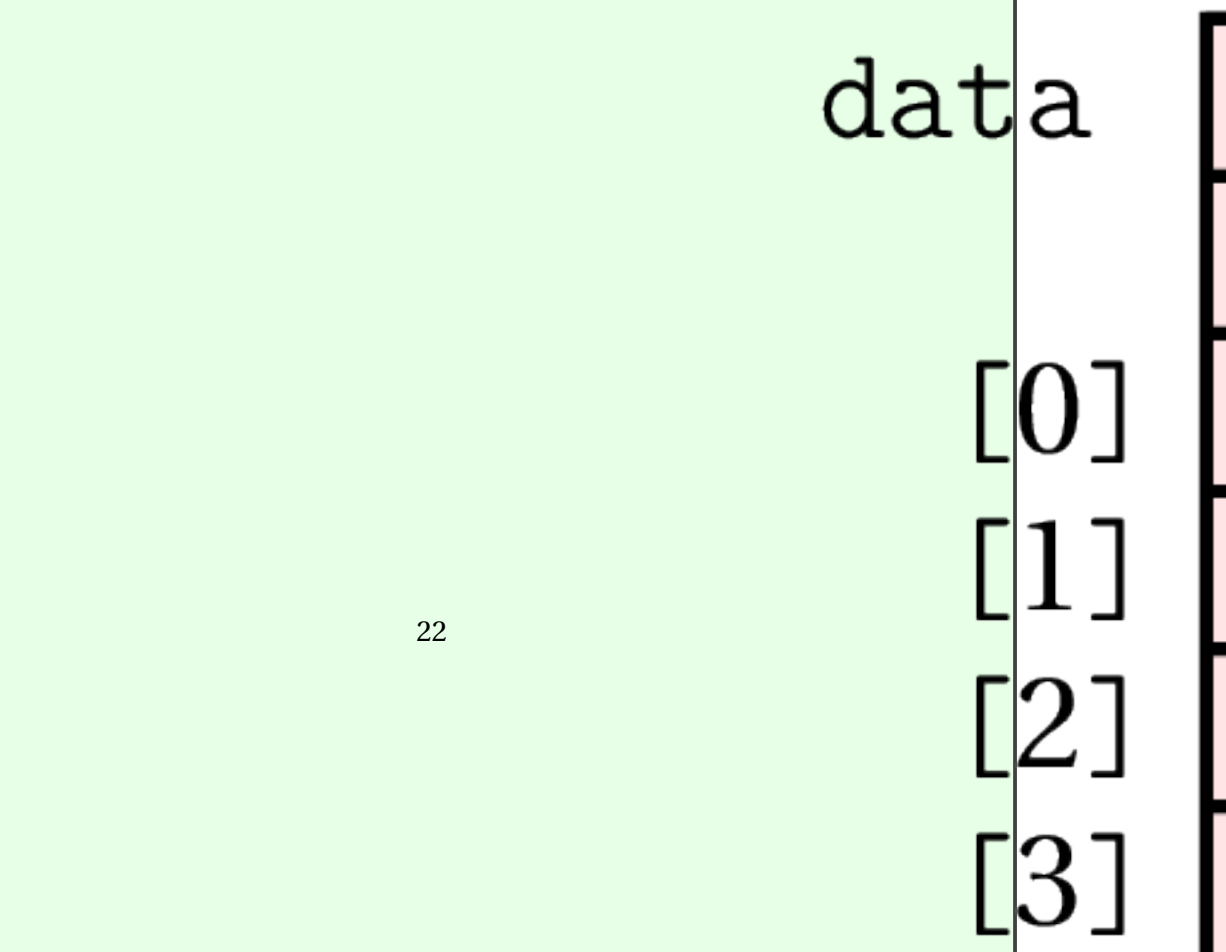
```
1 int item = arr[3];
```

We start off with the reference held by **arr** and getting the base address (600). To that we add 3*4 = 12 (3 being the index and 4 being size of each item), giving us the starting address of the item 612. We grab 4 bytes starting at that address (so bytes 612 to 615) and present it packed as an integer: 90.

Exercise 1

Show the steps involved in accessing item `data[5]` .

```
1 double[] data =          {12.5, 6.4, 8.5, 9.6, -3.4, -6.8, 0,
      -4.5};
```



To access an item at index **idx** , we just have to perform one set of operation, and grab x bytes from there (where x is the size of each item of the array).

The cost of this operation is fixed, irrespective of how big, or small, **idx** is. So, accessing an item in an ArrayList is $O(1)$ operation in both best and worst cases.

4.4.3 Inserting an item at a given index

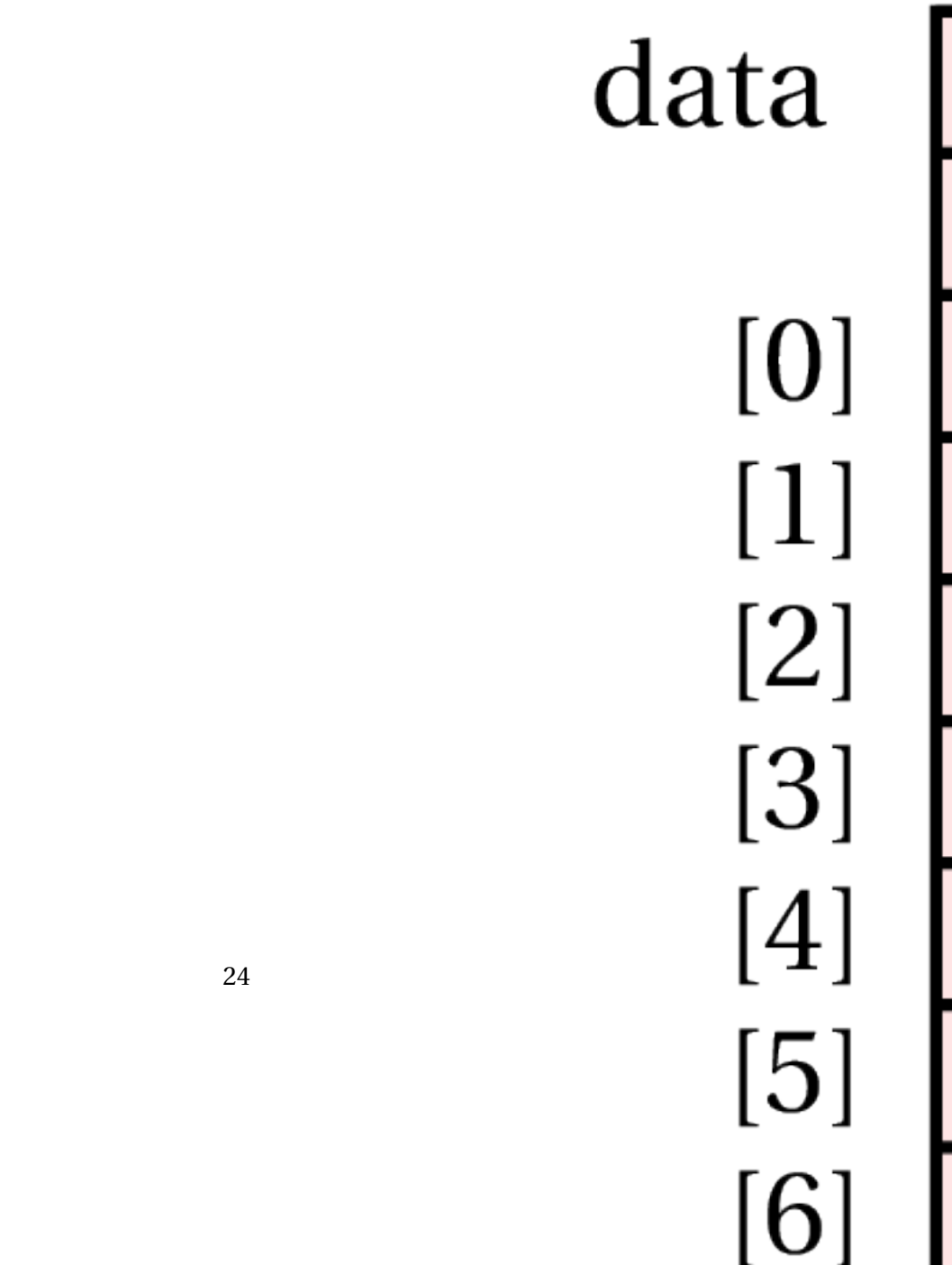
There are four cases for inserting an item:

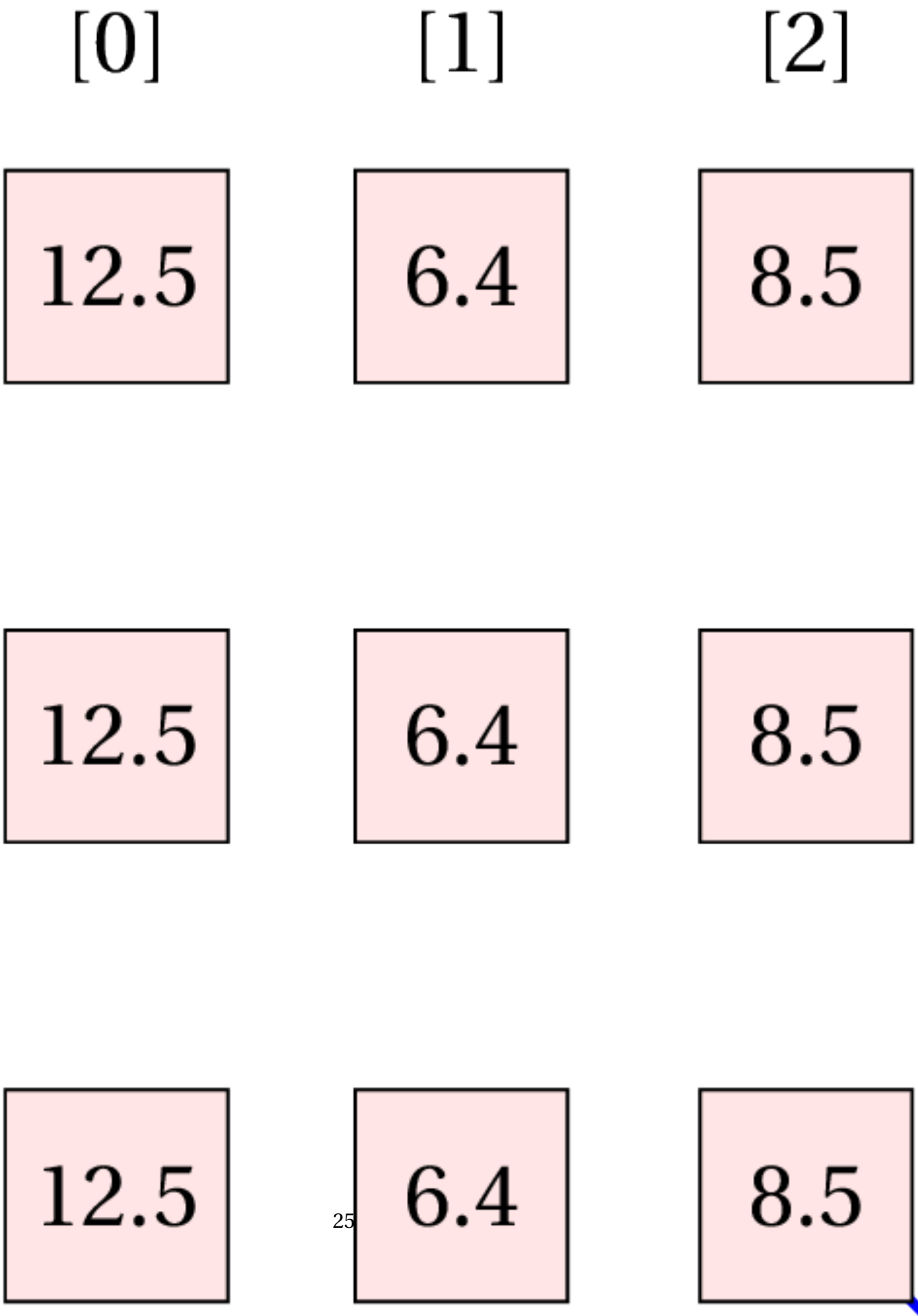
1. Array is not full and you are inserting an item at the end of the array: None of items need to be moved, array doesn't need to be grown. $O(1)$

2. Array is full and you are inserting an item at the end of the array: None of items need to be moved but array needs to be grown. Growing the array is $O(n)$. Therefore, time complexity in this case is $O(n)$.

3. Array is not full but you are inserting an item somewhere (not at the end): Some items need to be moved but array doesn't need to be grown. The worst case is when you are adding an item at index 0, in which case all items need to move, which is $O(n)$. Therefore, time complexity in this case is also $O(n)$.

   For example, if there are 5 items currently being used in an array of size 8, and we try to insert an item 5.6 at index 0, all 5 items need to be moved forward.


data
[0]
[1]
[2]
[3]
[4]
[5]
[6]


[0]
[1]
[2]
12.5
6.4
8.5
12.5
6.4
8.5
12.5
6.4
8.5

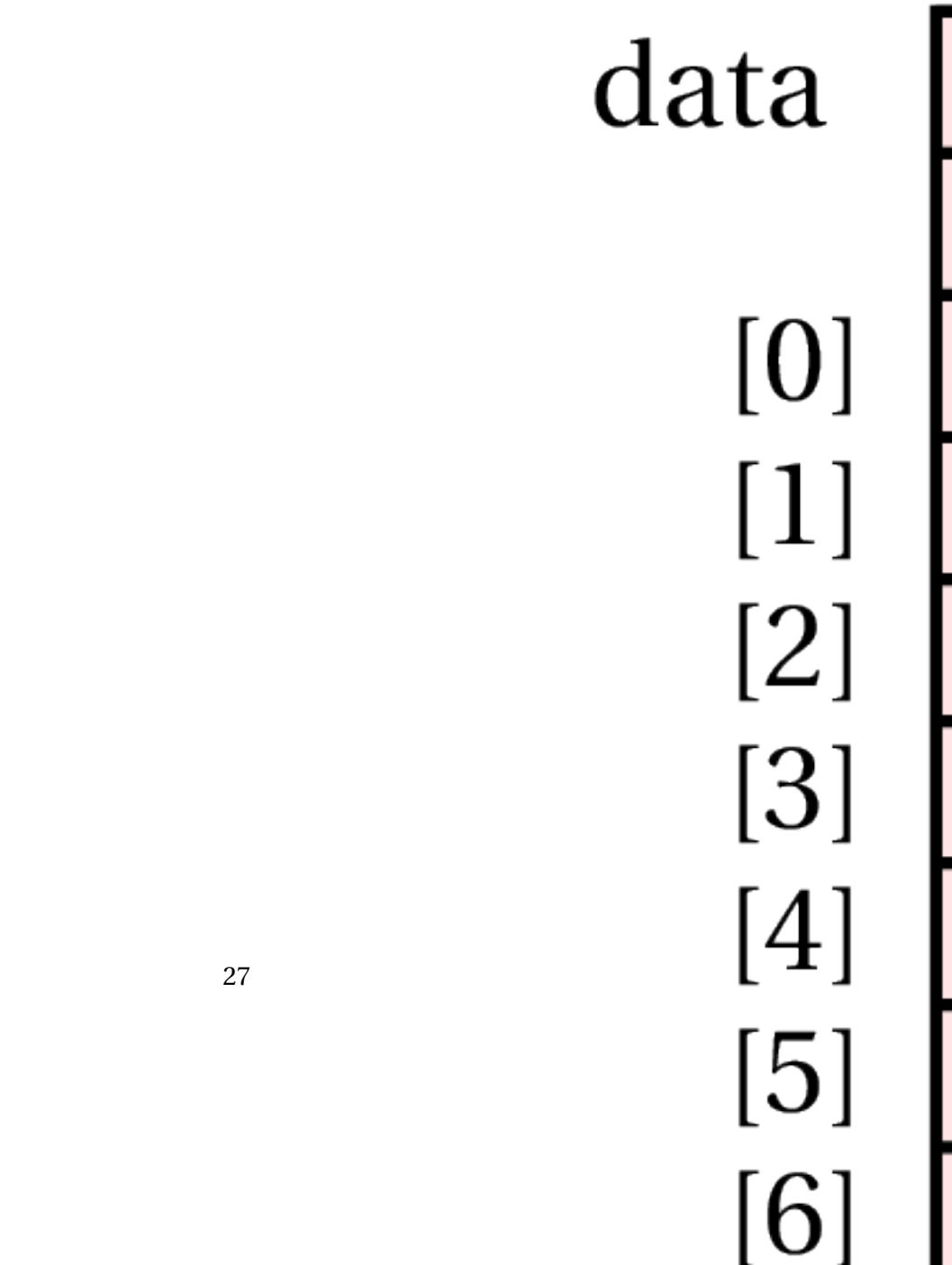
4. Array is full and you are inserting an item somewhere (not at the end): Double whammy, but $O(n) + O(n)$ is still $O(n)$.

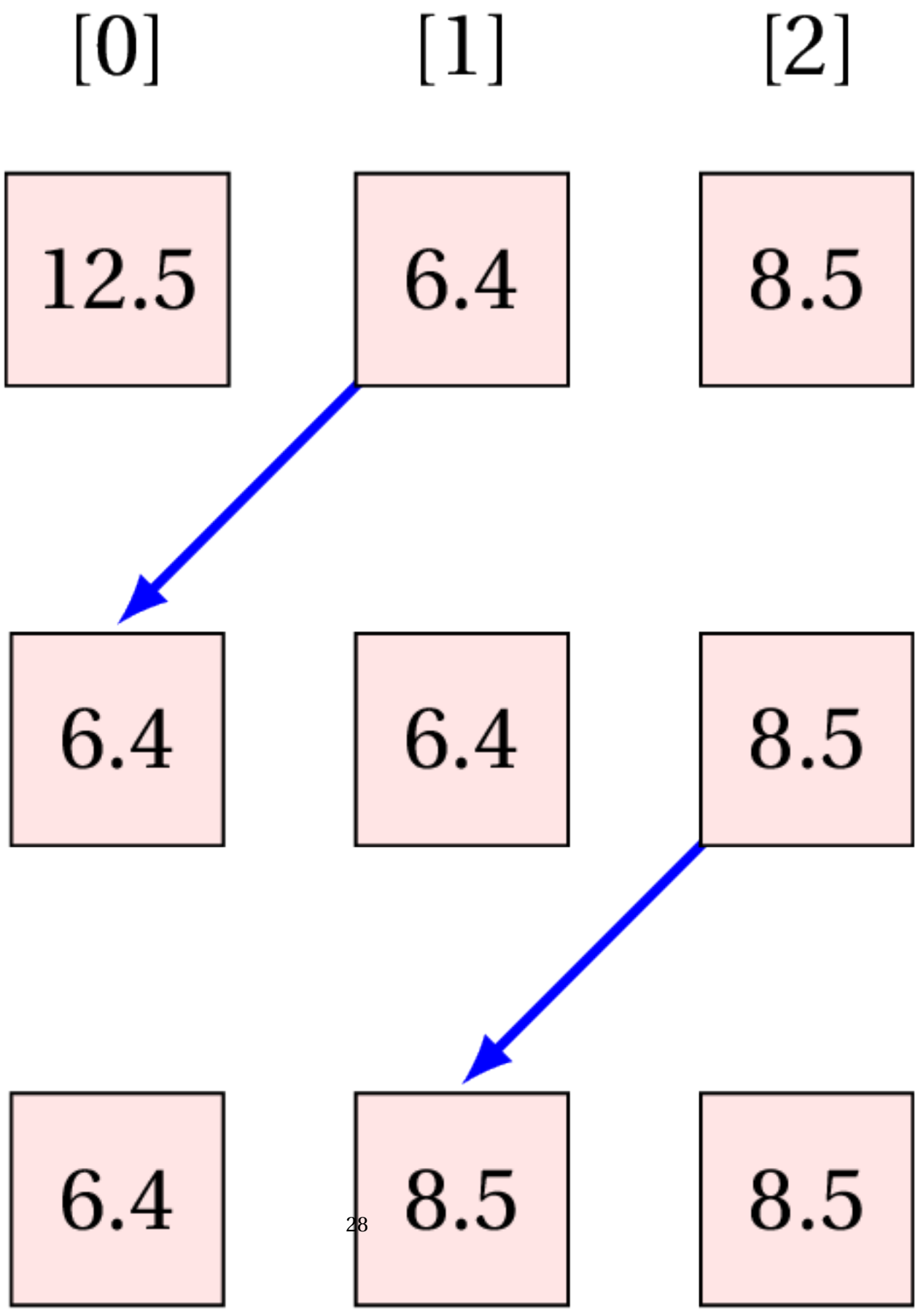
4.4.4 Removing an item at a given index

There are two relevant cases for removing an item:

1. Removing the last item: No items need to be moved. $O(1)$.
2. Removing the first item: All the items need to be moved. $O(n)$.

   For example, if there are 5 items currently being used in an array of size 8, and we try to remove the first item, all 5 items need to be moved backward by one.


data
[0]
[1]
[2]
[3]
[4]
[5]
[6]


[0]
[1]
[2]
12.5
6.4
8.5
6.4
6.4
8.5
6.4
8.5
8.5

Following table summarizes the time complexities of the various operations in an ArrayList:

Operation	Best case	Worst case
Accessing an item	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Inserting an item	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Removing an item	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Section 5.

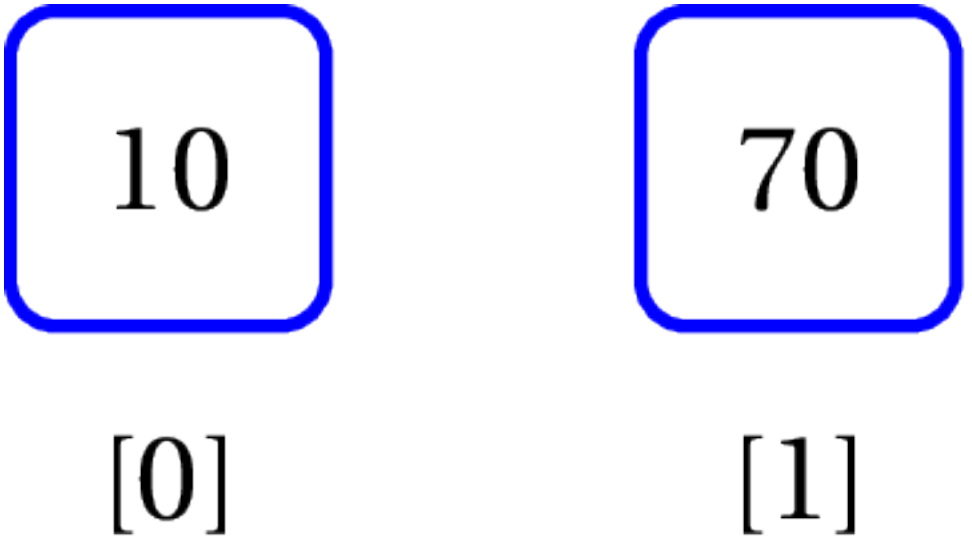
Iterators

Think of a bookmark that is placed between two pages such that the reader knows which page to read next.

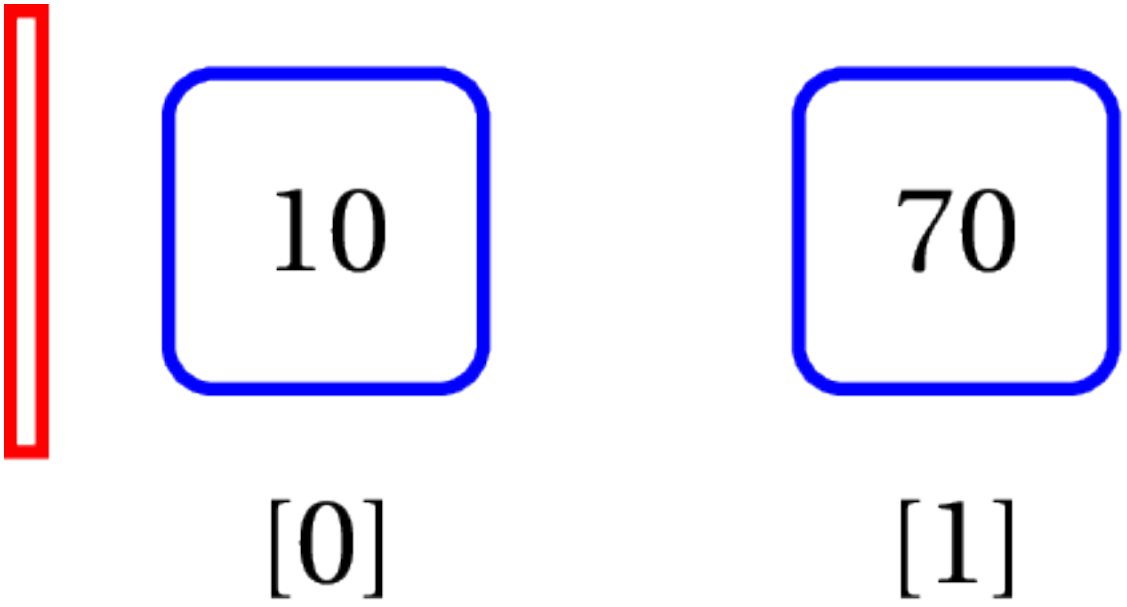
An iterator offers the exact same functionality.

1. check if there is any item left after the current position of the iterator, and if so, access it.
2. check if there is any item before the current iterator position and again, if so, access it.
3. remove the next item (if any).
4. change the value of the next item (if any).
5. insert an item after the current iterator position.

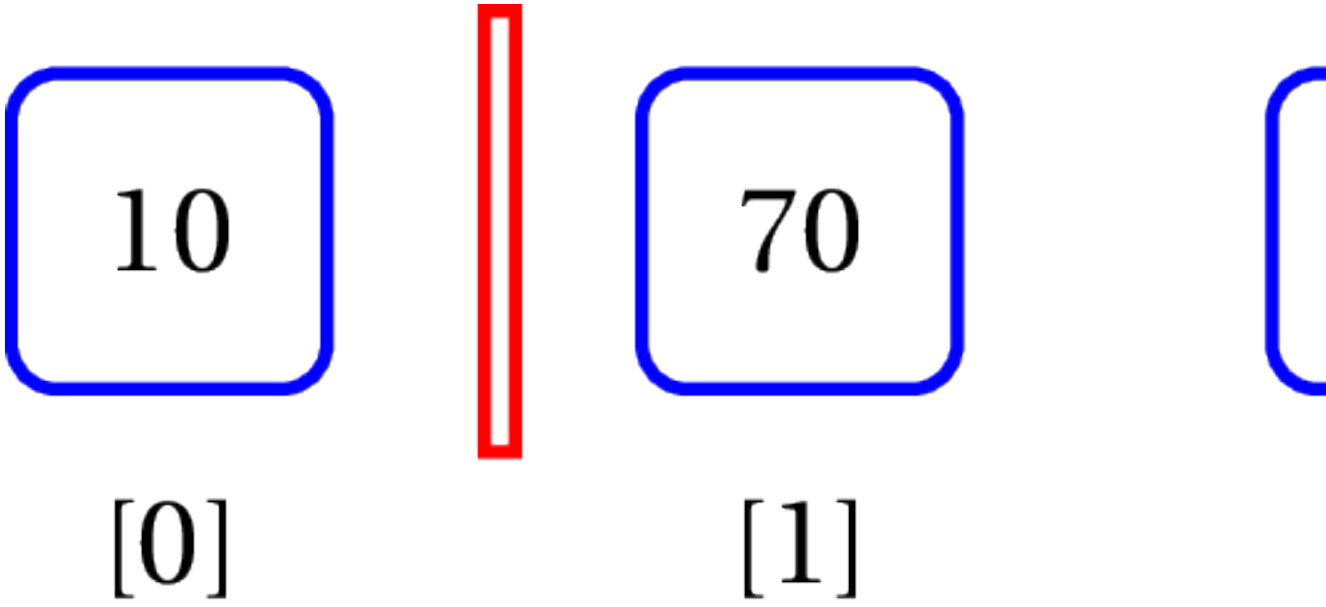
Consider a list containing items 10, 70, 20, 90, 30, 80


10
70
[0]
[1]

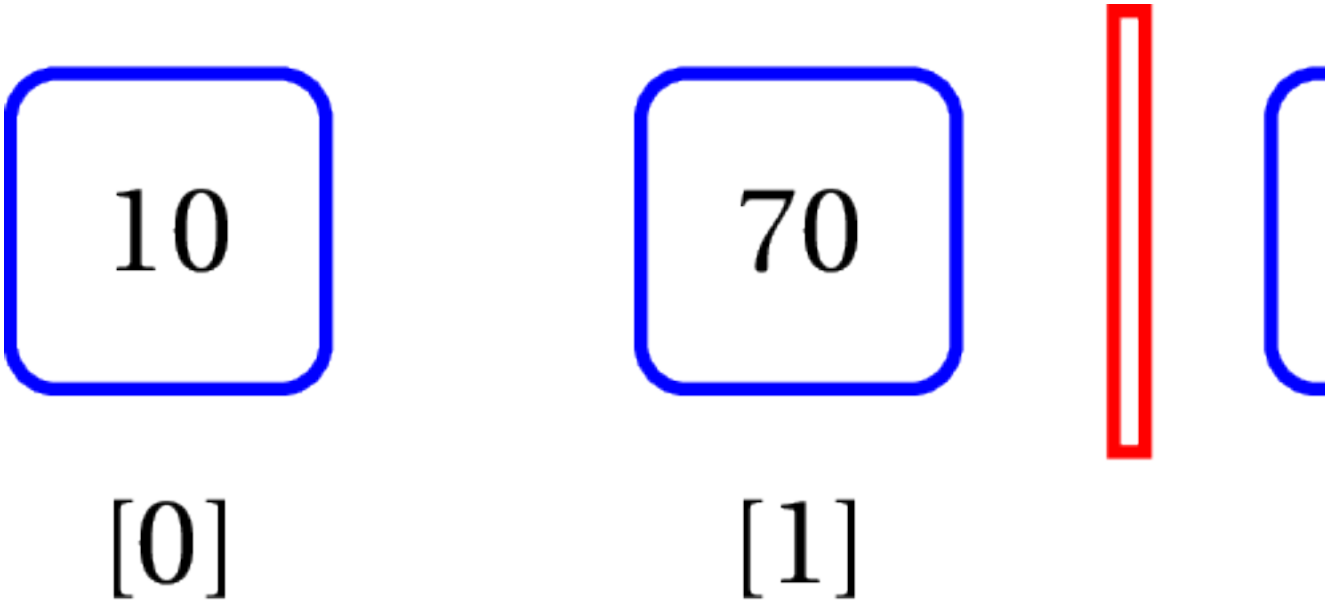
Initial state of the iterator:


10
70
[0]
[1]

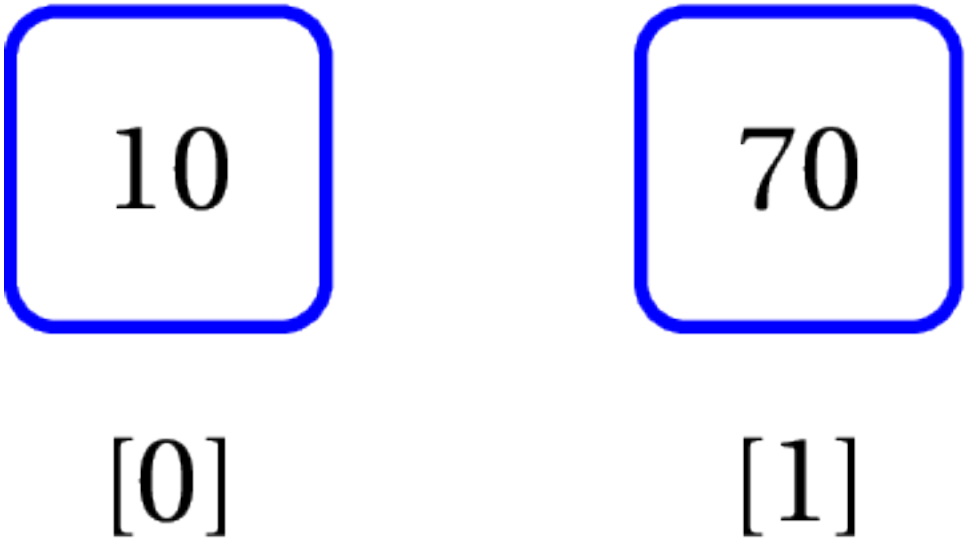
After accessing the first item (10):


10
70
[0]
[1]

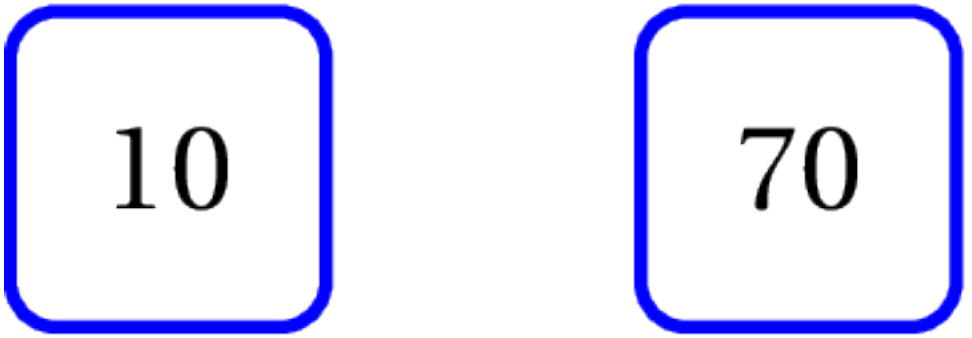
After accessing the second item (70):


10
70
[0]
[1]

After accessing the third item (20):

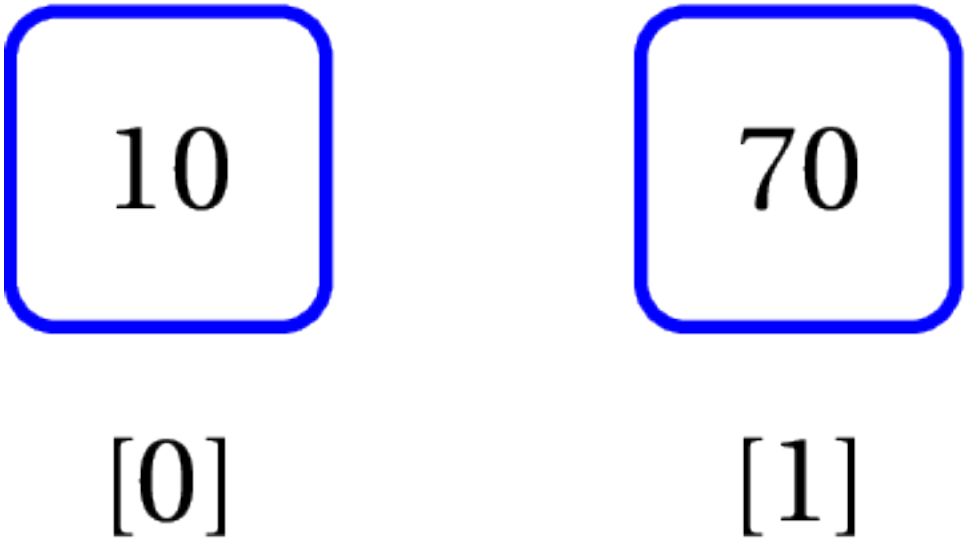

10
70
[0]
[1]

After accessing the fourth item (90):

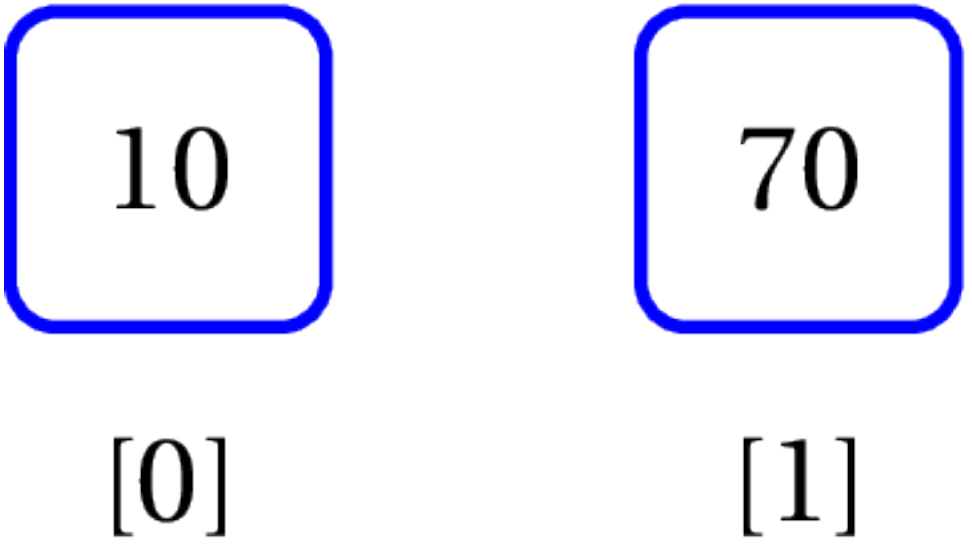


[0] [1]

After accessing the fifth item (30):


10
70
[0]
[1]

After accessing the last item (80):


10
70
[0]
[1]

5.1 Iterator implementation

An iterator is created for a collection and since we are dealing with lists, we'll be creating a **ListIterator** as:

```
Integer[] arr = {10,70,20,90,30,80};
ArrayList<Integer> list = new ArrayList(Arrays.asList(arr));

ListIterator<Integer> iter = list.listIterator();
```

You can check if there is an item "in front" of the iterator using **iter.hasNext()** which returns **true** if an items exists, **false** otherwise.

If an item exists, we can access it using **iter.next()** .

Thus, the traversal loop becomes,

```
while(iter.hasNext()) {
        System.out.print(iter.next()+"␣");
}
System.out.println();
```

You can also specify an index to the iterator from where it should begin.

```
Integer[] arr = {10,70,20,90,30,80};
ArrayList<Integer> list = new ArrayList(Arrays.asList(arr));
ListIterator<Integer> iter = list.listIterator(3);
//iterator is before item at index 3 (90)
```

Just like **hasNext(), next()** , you also have **hasPrevious(), previous()** .

In the example below, we start from the end of the list, iterating backwards.

```
Integer[] arr = {10,70,20,90,30,80};
ArrayList<Integer> list = new ArrayList(Arrays.asList(arr));

//start from the end
ListIterator<Integer> iter = list.listIterator(list.size());
while(iter.hasPrevious()) { //any item BEFORE the iterator?
        System.out.print(iter.previous()+"␣");
}
System.out.println();
```

Iterators can be passed to methods like any other object. The following method removes all even items from the list for which the iterator is created.

```
public static void removeEvens(ListIterator<Integer> iter) {
        while(iter.hasNext()) {
                if(iter.next() % 2 == 0) {
                        //we have moved forward
                        iter.previous(); //so, go back one space
                        iter.remove();
                }
        }
}
```

You can add an item before the item following the iterator using **add(Object)** . The iterator remains at the same item as before and not before the newly added item.

The following method adds a zero in front of all items in the list for which the iterator is created.

If the list was originally [10, 70, 20, 90], it would become [0, 10, 0, 70, 0, 20, 0, 90].

```
public static void padZeroes(ListIterator<Integer> iter) {
        while(iter.hasNext()) {
                iter.add(new Integer(0));
                iter.next();
        }
}
```

Section 6.

Sample solutions for exercises

Solution: Exercise 1

address of **data[5]** = base address of **data** + 5 * 8
= 320 + 40
= 360

data[5] = contents of memory from 360 to 367
= -6.8