Abstract Data Types

Intro to Abstract Data Types

The List Abstract Data Type

LinkedList vs. ArrayList

Abstract Data Types

A **data type** is a group of attributes and behaviours (i.e. an object).

An **abstract data type (ADT)** is a general data type, designed without a specific purpose in mind.

Advantages of **ADTs**:

They can be reused in many contexts.

Details about the implementation are abstracted away.

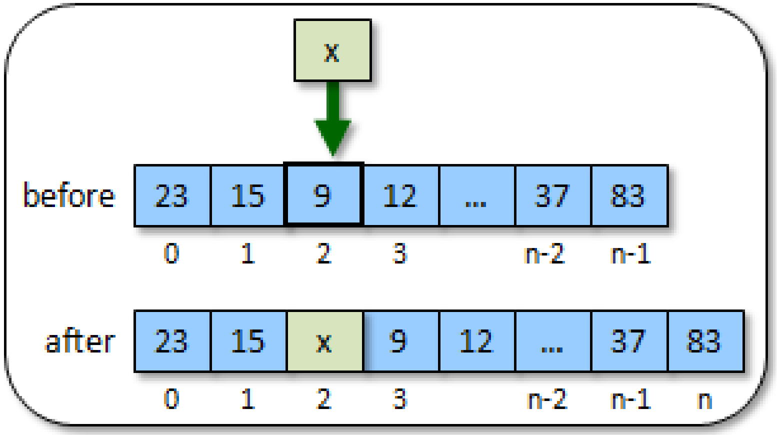
They take care of the details of the operations so you can focus on what you want to do with them.

Your code will be easier to understand!

The List ADT

A **list** is an abstract data type that implements an ordered collection of values, where the same value may occur more than once

```
add(int index, Object x)
aList.add(2, x)
```

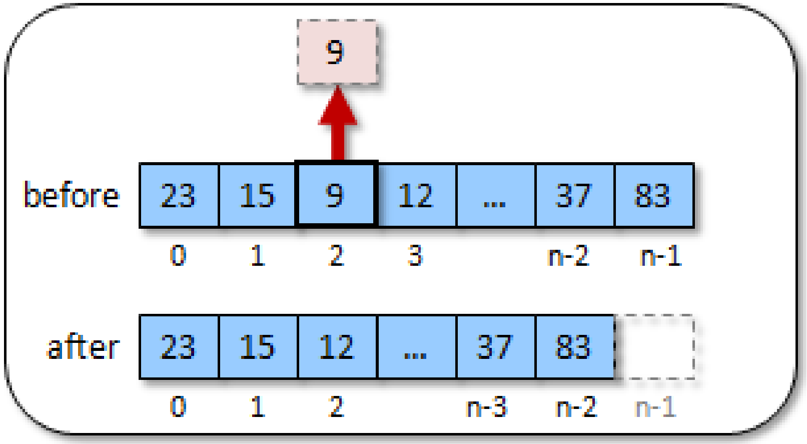


before	23	15	9	12	...	37	83
	0	1	2	3		n-2	n-1

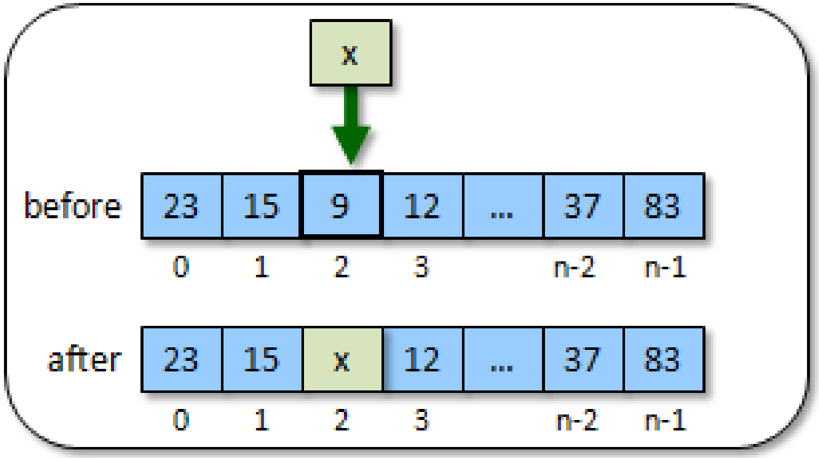
after	23	15	x	9	12	...	37	83
	0	1	2	3		n-2	n-1	n

`remove(int index)`

`aList.remove(2)` returns 9



```
set(int index, Object x)
aList.set(2, x)
```

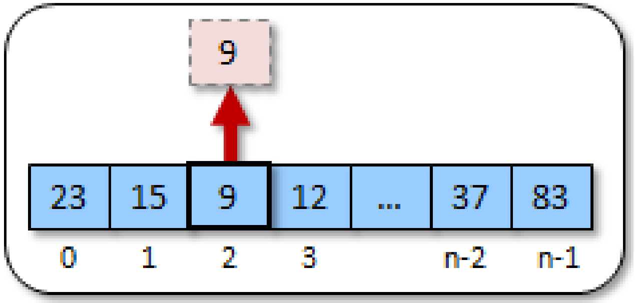


before 23 15 9 12 ... 37 83
0 1 2 3 n-2 n-1

after 23 15 x 12 ... 37 83
0 1 2 3 n-2 n-1

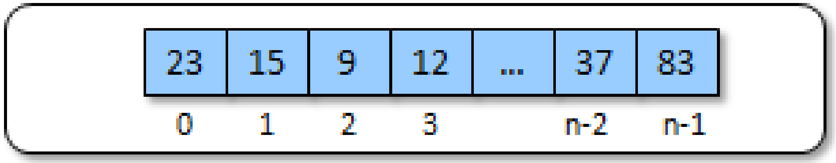
`get(int index)`

`aList.get(2)` returns 9

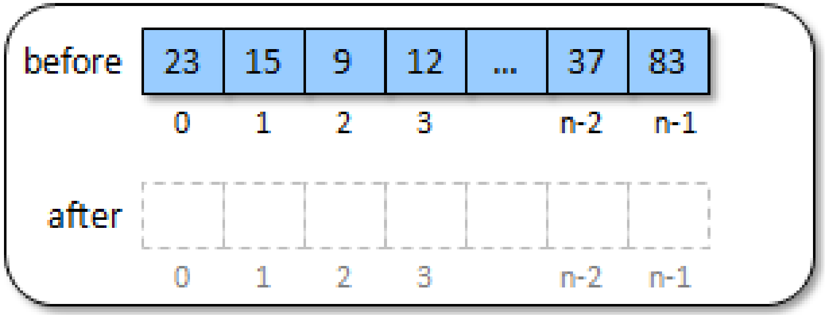


`size()`

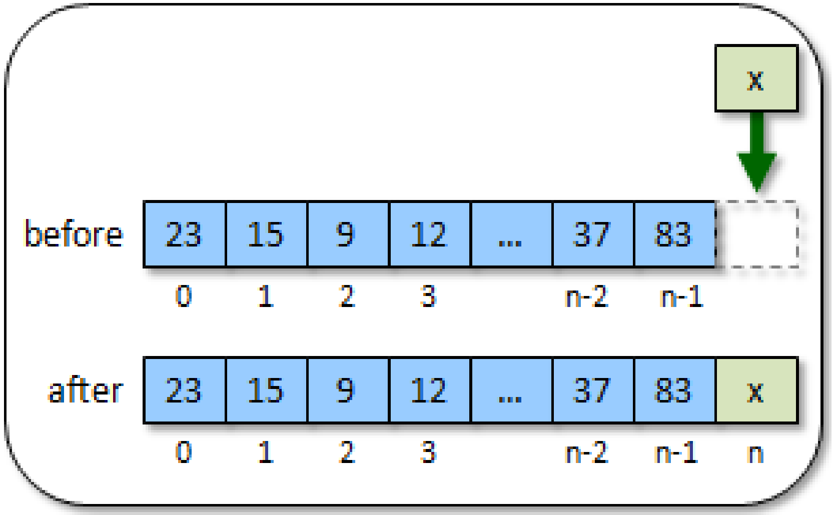
`aList.size()` returns n



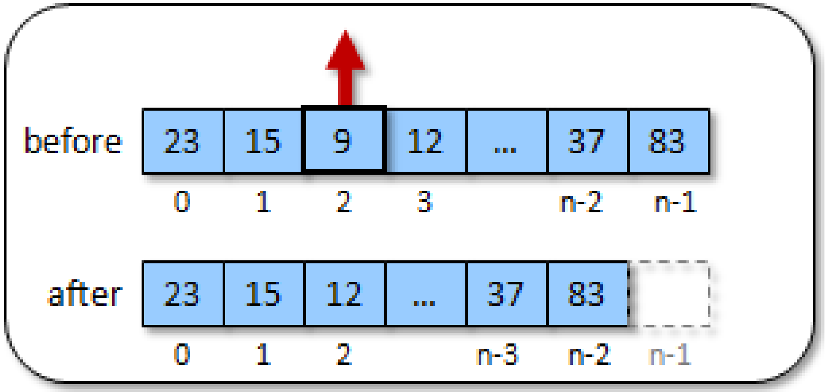
```
clear()
aList.clear()
```



```
add(Object x)
aList.add(x)
```

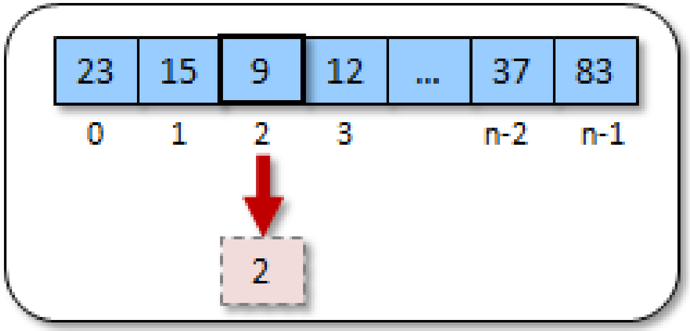


```
remove(Object x)
aList.remove(9)
```



`indexOf(Object x)`

`aList.indexOf(9)` returns 2



```
isEmpty()
```

same as

```
return (aList.size() == 0);
```

contains()

same as

```
for (int i=0; i<aList.size(); i++)
   if (aList.get(i).equals(x))
      return true;
return false;
```

The Java implementation of a list is called an `ArrayList`.

It is a fancy array whose messy details are abstracted away from you, the user.

```
import java.util.ArrayList;

public class ArrayListTestProgram
{
    public static void main(String[] args)
    {
        ArrayList myList;

        myList = new ArrayList();
        myList.add("Hello");
        myList.add(25);
        myList.add(new Person());
        myList.add(new Truck());
        System.out.println(myList);
    }
}
```

```
import java.util.ArrayList;

public clas
{
    public                                       args)
    {
        ArrayList myList;

        myList = new ArrayList();
        myList.add("Hello");
        myList.add(25);
        myList.add(new Person());
        myList.add(new Truck());
        System.out.println(myList);
    }
}
```

The Java implementation of a List

```
import java.util.ArrayList;

public class ArrayListTestProgram
{
    public static void main(String[] args)
    {
        ArrayList myList;

        myList = new ArrayList();
        myList.add("Hello");
        myList.add(25);
        myList.add(new Person());
        myList.add(new Truck());
        System.out.println(myList);
    }
}
```

Can add different kinds of objects (they actually get cast to `Object`)

To avoid warnings in previous code, specify what type of object we are putting into the list:

```
ArrayList<Object> myList;

myList = new ArrayList<Object>();
myList.add("Hello");
myList.add(25);
myList.add(new Person());
myList.add(new Truck());
System.out.println(myList);
```

Linked List vs Array List

ArrayList	Linked List
Elements are laid out contiguously in memory	Elements can be anywhere in memory
Getting an object is fast with indexing	Getting an object is slow with searching the list
Adding an element may involve shifting elements, or even moving to a new, bigger array	Adding an element involves updating references once the node is found
Removing an element may involve shifting elements	Removing an element involves updating references once the node is found