Introduction

Welcome to the List Section of the Java Masterclass.

In the last section, I talked about the array as a way to manage a list of items, all having the same type.

Arrays were a massive improvement if you needed to store items of the same type, but as you saw, Arrays have some limitations. Not being able to change the number of elements in an Array being one.

Fortunately, Java also includes an entire library for Java containers, which they call Collections.

Introduction

They take the arrays we worked with to the next level. They allow you to change the number of elements defined in an array for one, but there are many other improvements as well.

In this section, I'll be talking about lists, which are Java containers, and explain what they are and how to use them.

Two of the most common classes for lists are ArrayList and LinkedList. We'll start by looking at these.

Introduction

In addition, I'll be covering important concepts related to these topics like Big O Notation, Iterators, Autoboxing and Unboxing, and the enum type. If you are unfamiliar with any of these terms, by the end of this section, they will make sense.

There's a lot to cover in the section, so let's get started.

Java Array vs Java List

An array is mutable, and we saw that while we could set or change values in the array, we couldn't resize the array.

Java gives us several classes that let us add and remove items and resize a sequence of elements.

These classes are said to **implement** a List's behavior.

So, what is a list?

So what is a List?

In our everyday life, we use lists all the time.

When we're going to the grocery store, we've got a list.

We have a list of things we need to do, a list of addresses, a list of contact numbers, etc.

It wouldn't be a very useful list, however, if we started with 10 items that could be changed, but then couldn't add or remove an entry from that list.

So what is a List?

A List is a special type in Java, called an Interface.

For now, I'll say a List Interface describes a set of method signatures that all List classes are expected to have.

Let's look at some of these methods. I'm going to pull up the List methods in Java's API documentation page in a browser.

The ArrayList

The ArrayList is a class that maintains an array in memory that's actually bigger than what we need, in most cases.

It keeps track of the capacity or maximum size of the array in memory.

But it also keeps track of the elements that've been assigned or set, which is the size of the ArrayList.

As elements are added to an ArrayList, its capacity may need to grow. This occurs for you automatically, behind the scenes.

This is why the ArrayList is resizeable.

Arrays vs ArrayLists

This slide demonstrates that Arrays and ArrayLists have more in common than they don't.

Feature	Array	ArrayList
primitives types supported	Yes	**No**
indexed	Yes	Yes
ordered by index	Yes	Yes
duplicates allowed	Yes	Yes
nulls allowed	Yes, for non-primitive types	Yes
resizable	**No**	Yes
mutable	Yes	Yes
inherits from java.util.Object	Yes	Yes
implements List interface	No	**Yes**

Instantiating without Values

Instantiating Arrays	Instantiating ArrayLists
`String[] array = new String[10];`	`ArrayList<String> arrayList = new ArrayList<>();`
An array of 10 elements is created, all with null references. The compiler will only permit Strings to be assigned to the elements.	An empty ArrayList is created. The compiler will check that only Strings are added to the ArrayList.

On this slide, I show the differences when creating a new instance of an array compared to a new instance of an ArrayList.

An array requires square brackets in the declaration.

On the right-hand side of the equals sign, square brackets are also required with a size specified inside.

An ArrayList should be declared with the type of element for the ArrayList in angle brackets.

Instantiating without Values

Instantiating Arrays	Instantiating ArrayLists
`String[] array = new String[10];`	`ArrayList<String> arrayList = new ArrayList<>();`
An array of 10 elements is created, all with null references. The compiler will only permit Strings to be assigned to the elements.	An empty ArrayList is created. The compiler will check that only Strings are added to the ArrayList.

You can use the diamond operator when creating a new instance in a declaration statement.

You should use a specific type rather than just the Object class because Java can then perform compile-time type checking.

Instantiating with Values

Instantiating Arrays

```
String[] array = new String[] {"first", "second", "third"};
```

An array of 3 elements is created, with

```
elements[0] = "first"

elements[1] = "second"

elements[2] = "third
```

Alternately, we can use this array initializer (anonymous array).

```
String[] array = {"first", "second", "third"};
```

Instantiating Lists and Array Lists

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

An ArrayList can be instantiated by passing another list to it as we show here.

We can use the List.of() factory method, which uses variable arguments, to create a pass through immutable list.

You can use an array initializer to populate array elements during array creation.

This feature lets you pass all the values in the array as a comma delimited list in curly braces.

Instantiating with Values

Instantiating Arrays

```
String[] array = new String[] {"first", "second", "third"};
```

An array of 3 elements is created, with

```
elements[0] = "first"

elements[1] = "second"

elements[2] = "third
```

Alternately, we can use this array initializer (anonymous array).

```
String[] array = {"first", "second", "third"};
```

Instantiating Lists and Array Lists

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

An ArrayList can be instantiated by passing another list to it as we show here.

We can use the List.of() factory method, which uses variable arguments, to create a pass through immutable list.

When you use an array initializer in a declaration statement, you can use what's called the anonymous version, as I show here.

You can use an ArrayList constructor that takes a collection or a list of values during ArrayList creation.

Instantiating with Values

Instantiating Arrays

```
String[] array = new String[] {"first", "second", "third"};
```

An array of 3 elements is created, with

```
elements[0] = "first"

elements[1] = "second"

elements[2] = "third
```

Alternately, we can use this array initializer (anonymous array).

```
String[] array = {"first", "second", "third"};
```

Instantiating Lists and Array Lists

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

An ArrayList can be instantiated by passing another list to it as we show here.

We can use the List.of() factory method, which uses variable arguments, to create a pass through immutable list.

The List.of method can be used to create such a list, with a variable argument list of elements.

Element information

	Accessing Array Element data	Accessing ArrayList Element data
	Example `String[] arrays = {"first", "second", "third"};`	Example ArrayList: `ArrayList<String> arrayList = new ArrayList<>(List.of("first", "second", "third"));`
Index value of first element	`0`	`0`
Index value of last element	`arrays.length - 1`	`arrayList.size() - 1`
Retrieving number of elements:	`int elementCount = arrays.length;`	`int elementCount = arrayList.size();`
Setting (assigning an element)	`arrays[0] = "one";`	`arrayList.set(0, "one");`
Getting an element	`String element = arrays[0];`	`String element = arrayList.get(0);`

The number of elements is fixed when an array is created.

You can get the size of the array from the attribute length on the array instance.

Array elements are accessed with the use of square brackets and an index that ranges from 0 to one less than the number of elements.

Element information

	Accessing Array Element data	Accessing ArrayList Element data
	Example `String[] arrays = {"first", "second", "third"};`	Example ArrayList: `ArrayList<String> arrayList = new ArrayList<>(List.of("first", "second", "third"));`
Index value of first element	`0`	`0`
Index value of last element	`arrays.length - 1`	`arrayList.size() - 1`
Retrieving number of elements:	`int elementCount = arrays.length;`	`int elementCount = arrayList.size();`
Setting (assigning an element)	`arrays[0] = "one";`	`arrayList.set(0, "one");`
Getting an element	`String element = arrays[0];`	`String element = arrayList.get(0);`

The number of elements in an ArrayList may vary and can be retrieved with a method on the instance, named size().

ArrayList elements are accessed with get and set methods, also using an index ranging from 0 to one less than the number of elements.

Getting a String representation for Single Dimension Arrays and Arra

Array

Array Creation Code

```
String[] arrays = {"first", "second", "third"};
```

Printing Array Elements

```
System.out.println(Arrays.toString(arrays));
```

ArrayList

ArrayList Creation Code

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

Printing ArrayList elements

```
System.out.println(arrayList);
```

ArrayLists come with built-in support for printing out elements, including nested lists.

Arrays don't though, so you need to call Arrays.toString, passing the array as an argument.

This slide shows examples of single dimension arrays and ArrayLists.

Getting a String representation for Multi-Dimensional Arrays and Arr

Array

Array Creation Code

```
String[][] array2d = {

        {"first", "second", "third"},

        {"fourth", "fifth"}

};
```

Printing Array Elements

```
System.out.println(Arrays.deepToString(array2d));
```

ArrayList

ArrayList Creation Code

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

Printing ArrayList elements

```
System.out.println(multiDList);
```

Here, I show examples of multi-dimensional arrays and ArrayLists, and how to print the elements in each.

A multi-dimensional ArrayList simply has a type, which in itself is an ArrayList.

Getting a String representation for Multi-Dimensional Arrays and Arr

Array

Array Creation Code

```
String[][] array2d = {

        {"first", "second", "third"},

        {"fourth", "fifth"}

};
```

Printing Array Elements

```
System.out.println(Arrays.deepToString(array2d));
```

ArrayList

ArrayList Creation Code

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

Printing ArrayList elements

```
System.out.println(multiDList);
```

For a multi-dimensional array, you need to call the Arrays.deepToString method, passing the array as an argument.

For nested ArrayLists, we can still just pass the ArrayList instance directly, to System.out.println, as shown here.

Finding an element in an Array or ArrayList

Arrays methods for finding elements	ArrayList methods for finding elements

`int binarySearch(array, element)`
 	
** Array MUST BE SORTED
 	
Not guaranteed to return index of first element if there are duplicates
	`boolean contains(element)`
	
	`boolean containsAll(list of elements)`
	
	`int indexOf(element)`
	
	`int lastIndexOf(element)`

For arrays, you can use the binarySearch method to find a matching element, although this method requires that the array be sorted first.

In addition, if the array contains duplicate elements, the index returned from this search is not guaranteed to be the position of the first element.

For the ArrayList, we have several methods.

Finding an element in an Array or ArrayList

Arrays methods for finding elements	ArrayList methods for finding elements

`int binarySearch(array, element)`
 	
** Array MUST BE SORTED
 	
Not guaranteed to return index of first element if there are duplicates
	`boolean contains(element)`
	
	`boolean containsAll(list of elements)`
	
	`int indexOf(element)`
	
	`int lastIndexOf(element)`

You can use contains or containsAll, which simply returns a boolean if a match or matches were found.

In addition, like the String and StringBuilder, we have the methods, indexOf and lastIndexOf, which will return the index of the first or last match.

When a -1 is returned from these methods, no matching entry was found.

Sorting

Array	ArrayList

`String[] arrays = {"first", "second", "third"};`
 	
`Arrays.sort(arrays);`
	`ArrayList<String> arrayList = new ArrayList<>(List.of("first",`
	`"second", "third"));`
	
	`arrayList.sort(Comparator.naturalOrder());`
	`arrayList.sort(Comparator.reverseOrder());`

You can only sort arrays of elements that implement Comparable. We'll be discussing this in a future section. Character Sequence classes, like String and StringBuilder meet this requirement.	You can use the sort method with static factory methods to get Comparators.

Sorting seems like a simple concept when you think about sorting numbers and Strings.

We know there is a natural order for numbers and even for Strings.

Sorting

Array	ArrayList

`String[] arrays = {"first", "second", "third"};`
 	
`Arrays.sort(arrays);`
	`ArrayList<String> arrayList = new ArrayList<>(List.of("first",`
	`"second", "third"));`
	
	`arrayList.sort(Comparator.naturalOrder());`
	`arrayList.sort(Comparator.reverseOrder());`

You can only sort arrays of elements that implement Comparable. We'll be discussing this in a future section. Character Sequence classes, like String and StringBuilder meet this requirement.	You can use the sort method with static factory methods to get Comparators.

We can use the Arrays.sort method, for arrays with numeric primitive types and wrapper classes, as well as Strings and StringBuilders.

For the ArrayList, we can use the sort method again for numeric wrapper classes, Strings and StringBuilders.

Sorting

Array	ArrayList

`String[] arrays = {"first", "second", "third"};`
 	
`Arrays.sort(arrays);`
	`ArrayList<String> arrayList = new ArrayList<>(List.of("first",`
	`"second", "third"));`
	
	`arrayList.sort(Comparator.naturalOrder());`
	`arrayList.sort(Comparator.reverseOrder());`

You can only sort arrays of elements that implement Comparable. We'll be discussing this in a future section. Character Sequence classes, like String and StringBuilder meet this requirement.	You can use the sort method with static factory methods to get Comparators.

You pass a Comparator type argument to ArrayList's sort method that specifies how the sort should be performed.

You call static methods on the Comparator type to get a Comparator for either a natural order, or reverse order sort.

Array as an ArrayList

```
String[] originalArray = new String[] {"First", "Second", "Third"};
var originalList = Arrays.asList(originalArray);
```

There are times when you'll want to switch between an Array and an ArrayList, and there is support for this on both the Arrays class and the ArrayList class.

The Arrays.asList method returns an ArrayList backed by an array.

Here, I show the creation of a three element array.

Then the code uses the Arrays.asList method, passing it the array, and assigning the result, a List of Strings, to a variable, originalList.

Array as an ArrayList

```
String[] originalArray = new String[] {"First", "Second", "Third"};
var originalList = Arrays.asList(originalArray);
```

You can think of this conceptually, as putting an ArrayList wrapper of sorts around an existing array.

Any change made to the List is a change to the array that backs it.

This also means that an ArrayList created by this method is not resizable.

Creating Special Kinds of Lists

Using Arrays.asList	Using List.of
Returned List is NOT resizeable, but is mutable.	Returned List is IMMUTABLE.
`var newList = Arrays.asList("Sunday", "Monday", "Tuesday");`	`var listOne = List.of("Sunday", "Monday", "Tuesday");`
`String[] days = new String[] {"Sunday", "Monday", "Tuesday"};` `List<String> newList = Arrays.asList(days);`	`String[] days = new String[] {"Sunday", "Monday", "Tuesday"};` `List<String> listOne = List.of(days);`

This slide demonstrates two ways to create a list. From elements or from an array of elements.

Both are static factory methods on types.

The first is the asList method on the Arrays class, and it returns a special instance of a List that is not resizable, but is mutable.

The second is the of method on the List interface, and it returns a special instance of a List that is immutable.

Creating Special Kinds of Lists

Using Arrays.asList	Using List.of
Returned List is NOT resizeable, but is mutable.	Returned List is IMMUTABLE.
`var newList = Arrays.asList("Sunday", "Monday", "Tuesday");`	`var listOne = List.of("Sunday", "Monday", "Tuesday");`
`String[] days = new String[] {"Sunday", "Monday", "Tuesday"};` `List<String> newList = Arrays.asList(days);`	`String[] days = new String[] {"Sunday", "Monday", "Tuesday"};` `List<String> listOne = List.of(days);`

Both support variable arguments, so you can pass a set of arguments of one type, or you can pass an array.

I am showing examples of both here, first using variable arguments, and second, passing an array.

Creating Arrays from ArrayLists

```
ArrayList<String> stringLists = new ArrayList<>(List.of("Jan","Feb","Mar"));
String[] stringArray = stringLists.toArray(new String[0]);
```

This slide shows the most common method to create an array from an ArrayList using the method toArray.

This method takes one argument which should be an instance of a typed array.

This method returns an array of that same type.

If the length of the array you pass has more elements than the list, extra elements will be filled with the default values for that type.

If the length of the array you pass has less elements than the list, the method will still return an array, with the same number of elements in it, as the list.

In the example shown here, I pass a String array with zero as the size, but the array returned has three elements, which is the number of elements in the list.

Array of primitive values

When an array of primitive types is allocated, space is allocated for all of it's elements contiguously, as shown here.

You can see from this slide that we have an array of **seven integers**.

The index position is in the left column, and that's the number we use to access a specific array value.

So the first element, when we use index position 0, this will retrieve the value 34.

When we use index position 1, this gets the value of 18, and so on.

The addresses I show here are memory addresses represented by these numbers.

Index	Value	Address
0	34	100
1	18	104
2	91	108
3	57	112
4	453	116
5	68	120
6	6	124

Array of primitive values

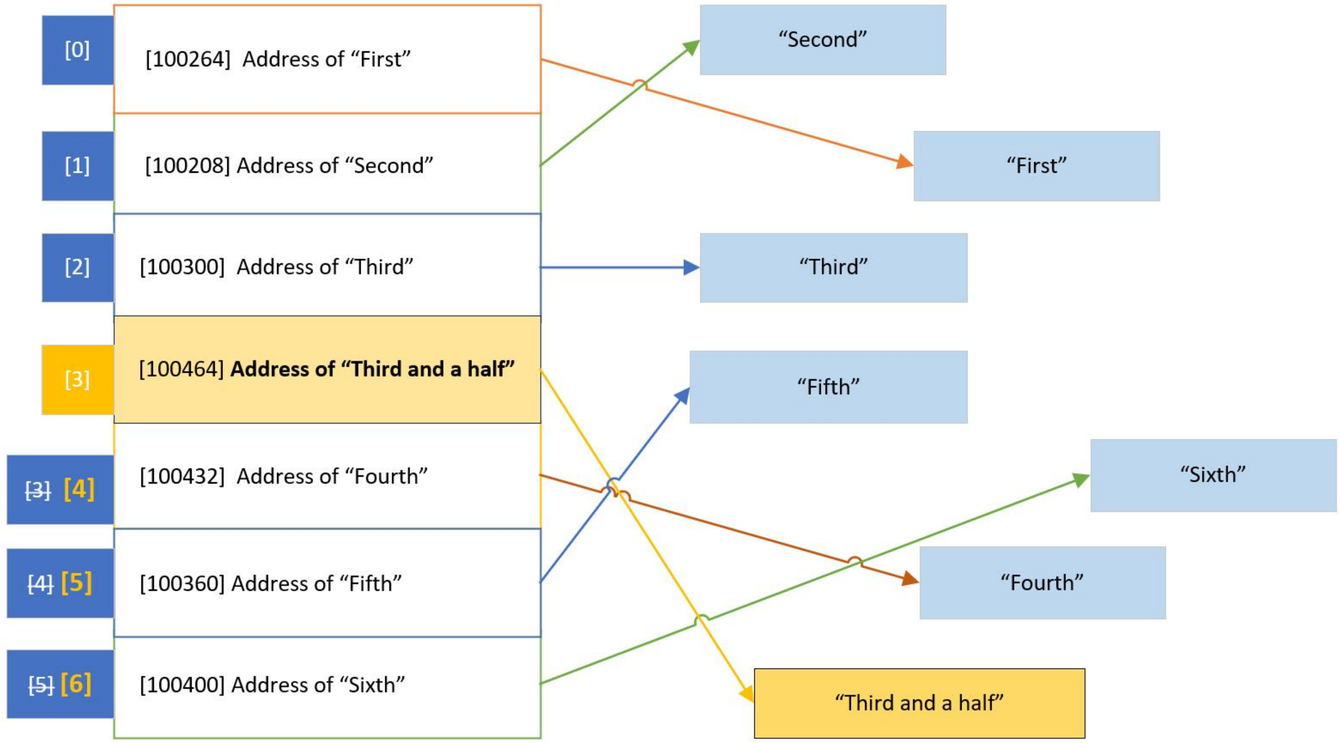
If 100 is the address of an integer and we know an integer is 4 bytes, then the address of the next integer if it's contiguous would be 104, as I show here for the second element.

Java can use simple math using the index and the address of the initial element in the array to get the address and retrieve the value of the element.

Index	Value	Address
0	34	100
1	18	104
2	91	108
3	57	112
4	453	116
5	68	120
6	6	124

Arrays and ArrayLists of reference types

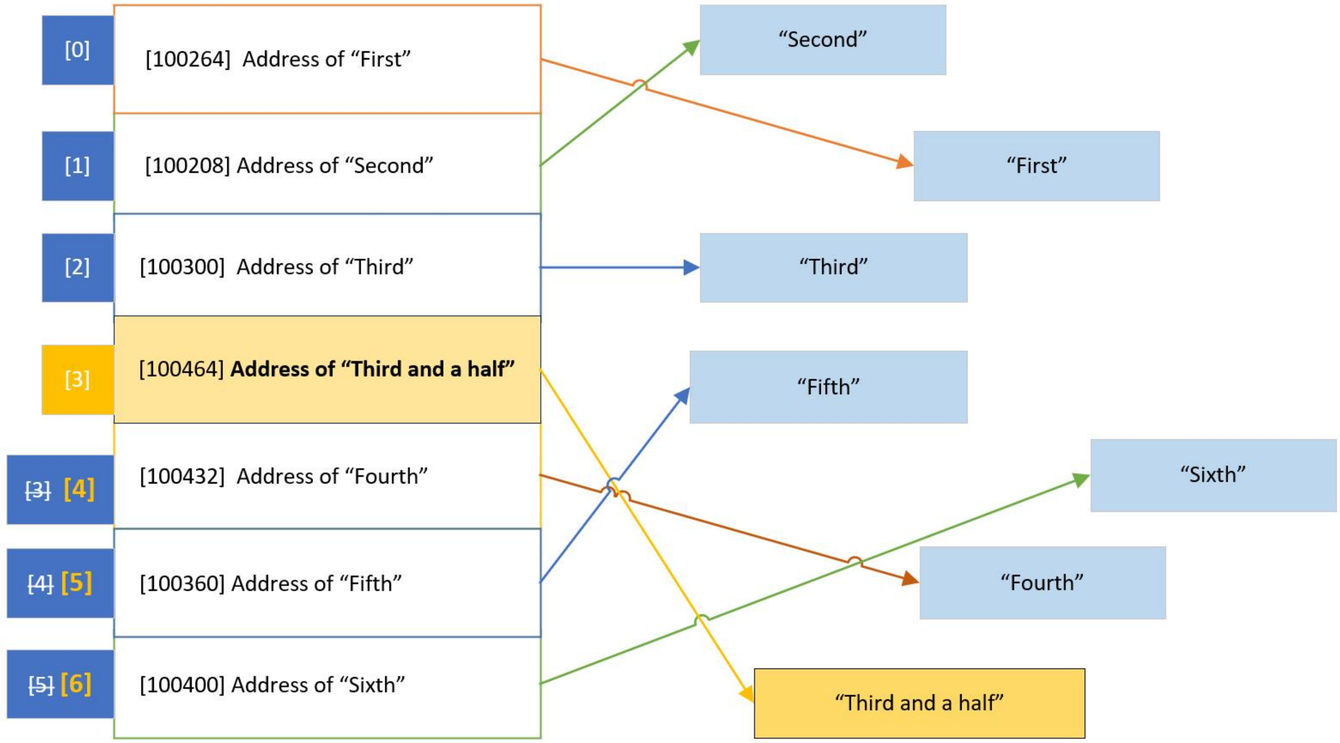
For reference types (meaning anything that's not a primitive type) like a String or any other object, the array elements aren't the values, but the addresses of the referenced object or String. I've talked about this before.



Arrays and ArrayLists of reference types

There's a level of indirection as I show on this slide.

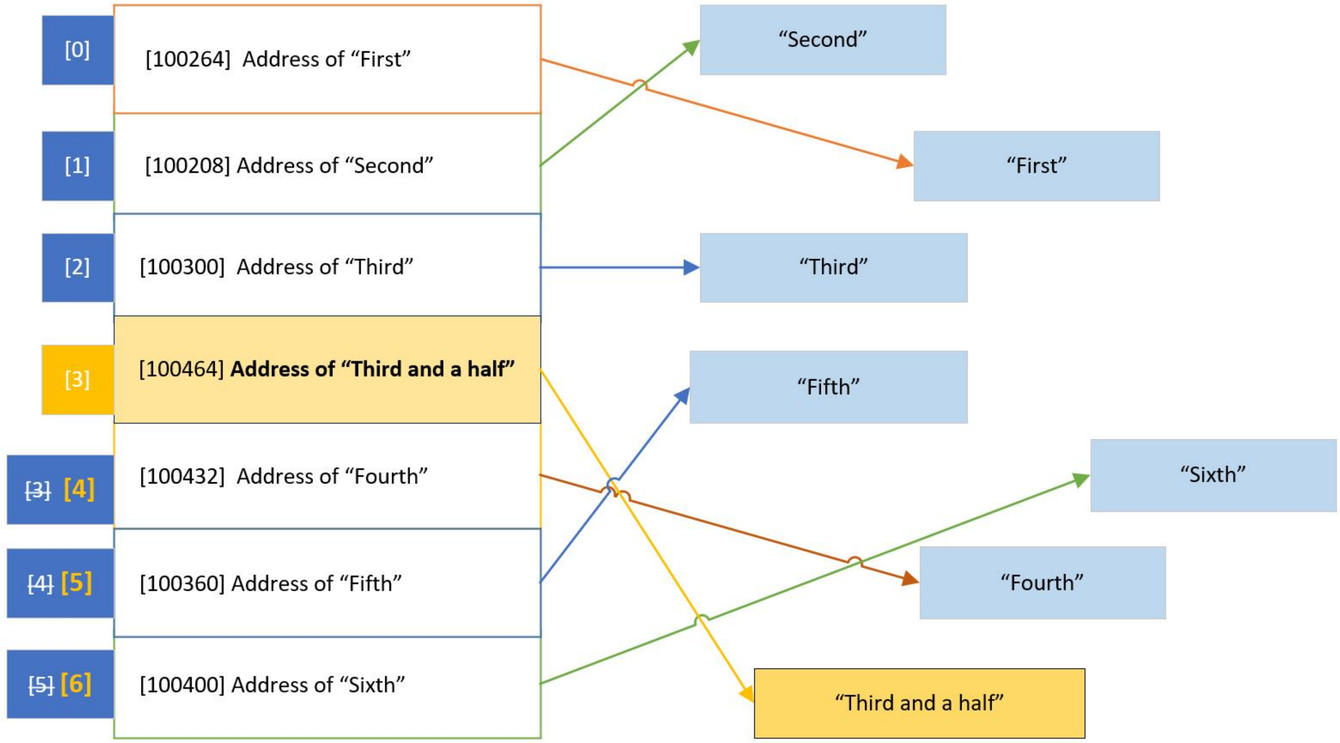
You've learned that ArrayLists are really implemented with arrays under the covers.



Arrays and ArrayLists of reference types

This means our objects aren't stored contiguously in memory, but their addresses are, in the array behind the ArrayList.

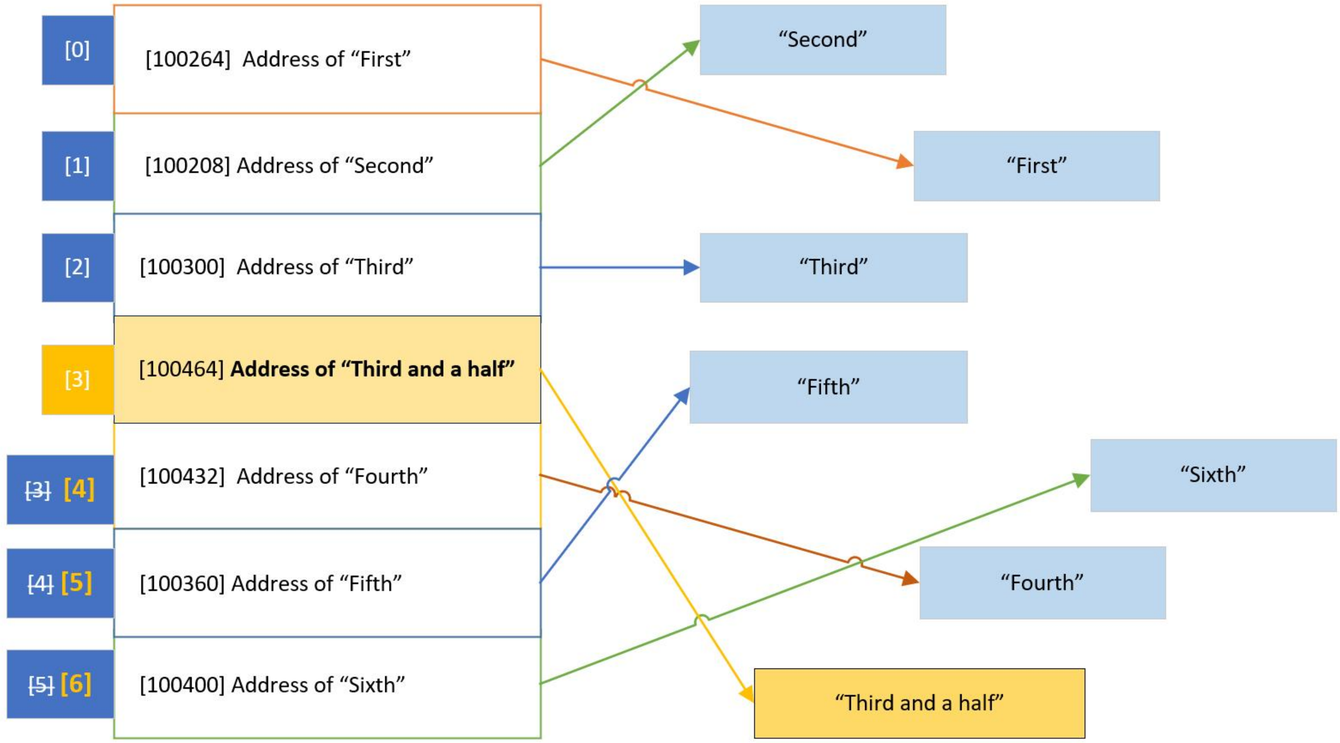
And again, the addresses can be easily retrieved with a bit of math, if we know the index of the element.



Arrays and ArrayLists of reference types

This is a **cheap** or fast lookup and doesn't change no matter what size the ArrayList is.

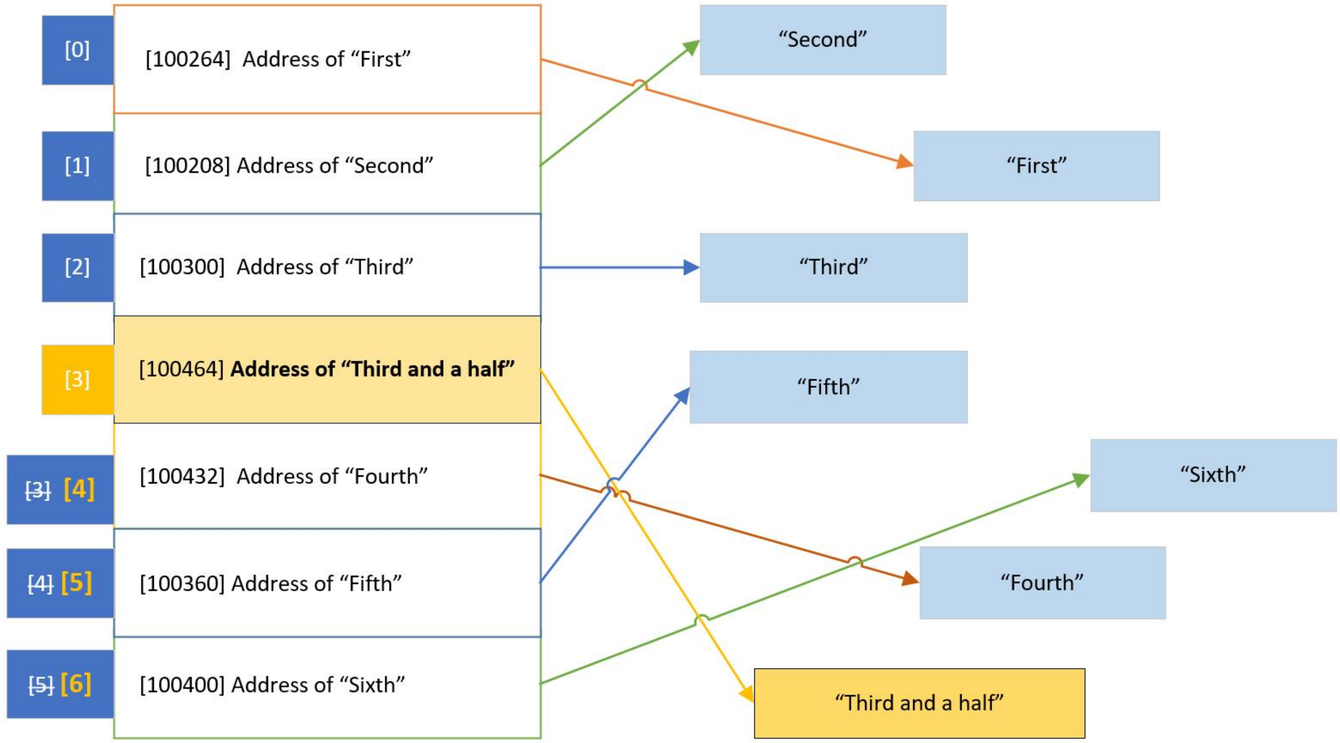
But to remove an element, the referenced addresses have to be re-indexed or shifted to remove an empty space.



Arrays and ArrayLists of reference types

And when adding an element, the array that backs the ArrayList might be too small and might need to be reallocated.

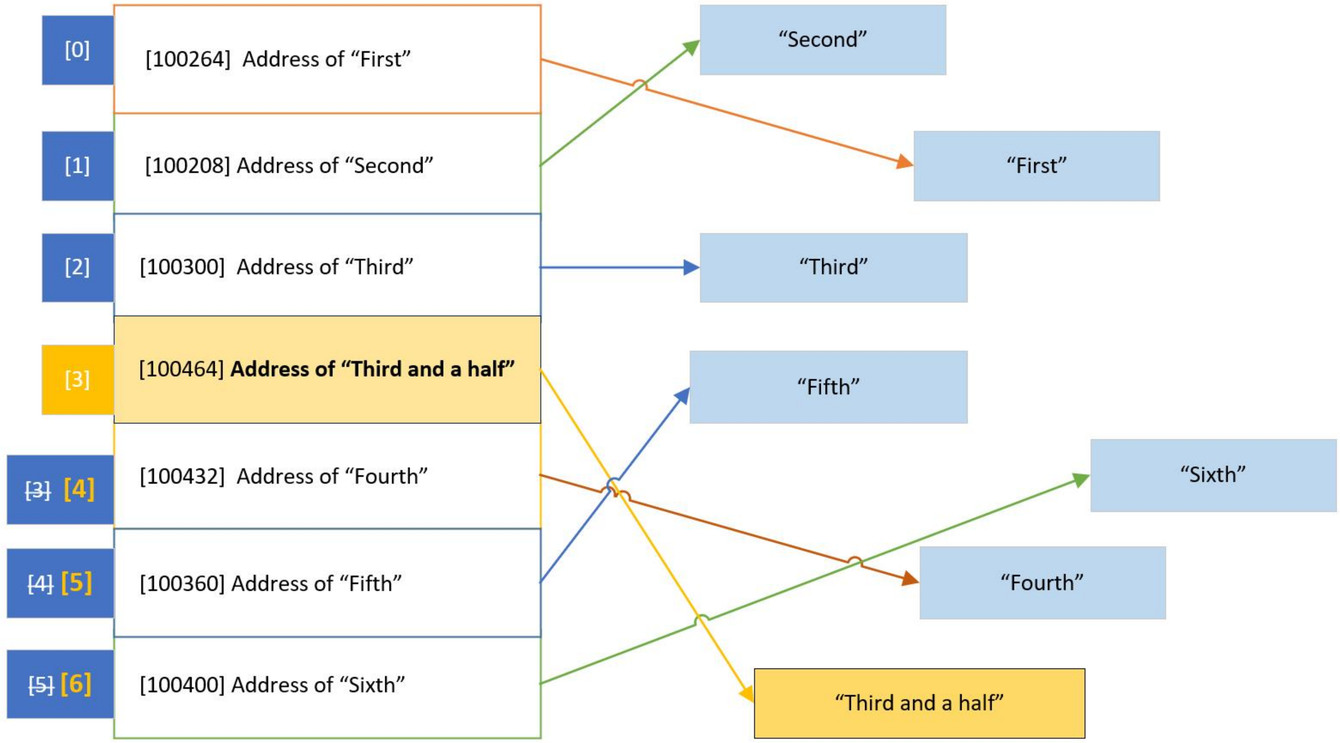
Either of these operations can be an **expensive** or time-consuming process if the number of elements is large.



Arrays and ArrayLists of reference types

In this slide, the String, "Third and a half', represents a new element I want inserted at index position 3.

This means all the elements below this point need to be moved and re-indexed.



ArrayList capacity

An ArrayList is created with an initial capacity depending on how many elements we create the list with, or if you specify a capacity when creating the list.

On this slide, I show an ArrayList that has a capacity of 10 because I'm passing 10 in the constructor of this list.

I then add 7 elements.

```
ArrayList<Integer> intList = new ArrayList<>(10);
for (int i = 0; i < 7; i++) {
    intList.add((i + 1) * 5);
}
```

Index	0	1	2	3	4	5	6	7	8	9
Value	5	10	15	20	25	30	35			

ArrayList capacity

I can add 3 more elements using the ArrayList add method, and the array that is used to store the data doesn't need to change.

```
ArrayList<Integer> intList = new ArrayList<>(10);
for (int i = 0; i < 7; i++) {
    intList.add((i + 1) * 5);
}

intList.add(40);
intList.add(45);
intList.add(50);
```

The elements at indices 7, 8, and 9 get populated.

Index	0	1	2	3	4	5	6	7	8	9
Value	5	10	15	20	25	30	35	40	45	50

ArrayList capacity is reached

But if the number of elements exceeds the current capacity, Java needs to reallocate memory to fit all the elements, and this can be a costly operation, especially if your ArrayList contains a lot of items.

```
ArrayList<Integer> intList = new ArrayList<>(10);
for (int i = 0; i < 7; i++) {
    intList.add((i + 1) * 5);
}
intList.add(40);
intList.add(45);
intList.add(50);

intList.add(55);              // This add exceeds the ArrayList capacity,
                              // assuming an initial capacity of 10,
                              // as an example.
```

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value	5	10	15	20	25	30	35	40	45	50	55				

ArrayList capacity is reached

So now, if our code simply calls add on this ArrayList, the next operation is going to create a new array, with more elements, but needs to copy the existing 10 elements over.

Then the new element is added. You can imagine this add operation costs more, in both time and memory, than the previous add methods did.

When Java reallocates new memory for the ArrayList, it automatically sets the capacity to a greater capacity.

But the Java language doesn't really specify exactly how it determines the new capacity or promise that it will continue to increase the capacity in the same way in future versions.

We can't actually get this capacity size from the ArrayList.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value	5	10	15	20	25	30	35	40	45	50	55				

ArrayList capacity is reached

From their own documentation, Java states that, "The details of the growth policy are not specified beyond the fact that adding an element, has constant amortized time cost".

Ok, maybe you're interested in what constant amortized time is.

Let's start with how to determine cost, which in this case is generally considered in terms of time, but may include memory usage and processing costs, etc.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value	5	10	15	20	25	30	35	40	45	50	55				

Big O Notation

Maybe you've heard people talking about Big O Notation, or Big O, and wondered what this means.

I won't get too deep into it, but there are a couple of concepts that are fairly easy to grasp, and will help us understand how **cheap** or **expensive** an operation is in terms of time and memory usage, as the operation scales.

This means it's a way to express how well the operation performs when applied to more and more elements.

Big O approximates the cost of an operation for a certain number of elements called n.

Cost is usually determined by the time it takes, but it can include memory usage and complexity, for example.

Big O Notation

As n (the number of elements) gets bigger, an operation's cost can stay the same.

But cost often grows as the number of elements grow.

Costs can grow linearly, meaning the cost stays in step with the magnitude of the number of elements.

Or costs can grow exponentially or by some other non-linear fashion.

In a perfect world, an operation's time and complexity would never change. This ideal world, in Big O Notation is O(1), sometimes called constant time.

In many situations, an operation's cost is in direct correlation to the number of elements, n. In Big O Notation this is O(n), sometimes called linear time.

Big O Notation

So, if we have 10 elements, the cost is 10 times what it would be for 1 element because the operation may have to execute some functions up to 10 times vs. just once, and 100 times for 100 elements, for example.

O(n) is generally our worst case scenario for List operations, but there are Big O Notations, for worse performers.

Constant Amortized Time Cost

Another scenario is the one the Java docs declared for the growth of the ArrayList that adding an element has constant amortized time cost.

In our case, we'll designate this constant amortized time as O(1)*.

This means that in the majority of cases, the cost is close to O(1), but at certain intervals, the cost is O(n).

If we add an element to an ArrayList where the capacity of the List is already allocated and space is available, the cost is the same each time, regardless of how many elements we add.

Constant Amortized Time Cost

But as soon as we reach the capacity and all the elements (all n elements) need to be copied in memory, this single add would have a maximum cost of O(n).

After this operation, that forced a reallocation, any additional add operations go back to O(1), until the capacity is reached again.

As the expensive intervals decrease, the cost gets closer to O(1), so we give it the notation O(1)*.

ArrayList Operations - Big O

This slide shows the Big O values for the most common ArrayList operations or methods.

Let's just talk about one example, the contains method, which looks for a matching element, and needs to traverse through the ArrayList to find a match.

It could find a match at the very first index, this is the best case scenario, so it's O(1).

It might not find a match until the last

Operation	Worst Case	Best Case
add(E element)	O(1)*	
add(int index, E element)	O(n)	O(1)*
contains(E element)	O(n)	O(1)
get(int index)	O(1)	
indexOf(E Element)	O(n)	O(1)
remove(int index)	O(n)	O(1)
remove(E element)	O(n)	
set(int index, E element)	O(1)	

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

ArrayList Operations - Big O

In general, the cost will be something in between for the contains method because the element will be found somewhere between the first and nth (or last) element.

You'll notice that the indexed methods are usually O(1), remembering that finding an element by its index is a simple calculation.

It only gets costly with indexed add or remove methods, if the ArrayList needs to

Operation	Worst Case	Best Case
add(E element)	O(1)*	
add(int index, E element)	O(n)	O(1)*
contains(E element)	O(n)	O(1)
get(int index)	O(1)	
indexOf(E Element)	O(n)	O(1)
remove(int index)	O(n)	O(1)
remove(E element)	O(n)	
set(int index, E element)	O(1)	

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

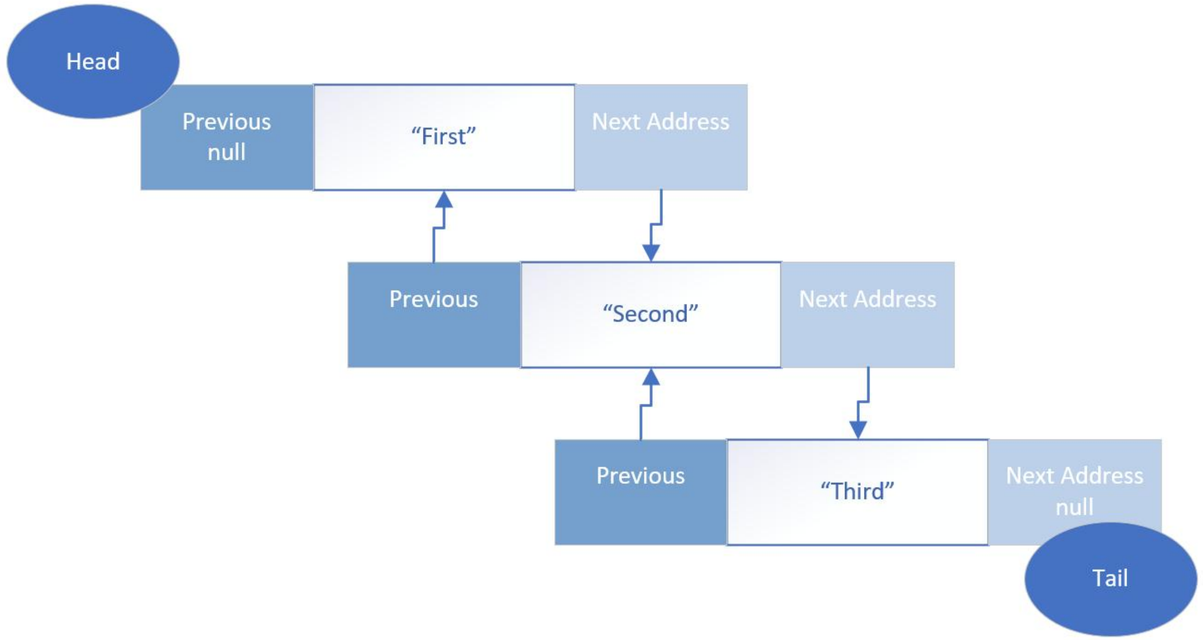
O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

LinkedList

The LinkedList is not indexed at all.

There is no array storing the addresses in a neat, ordered way, as we saw with the ArrayList.

Instead, each element that's added to a linked list forms a chain and the chain has links to the previous element, and the next element.

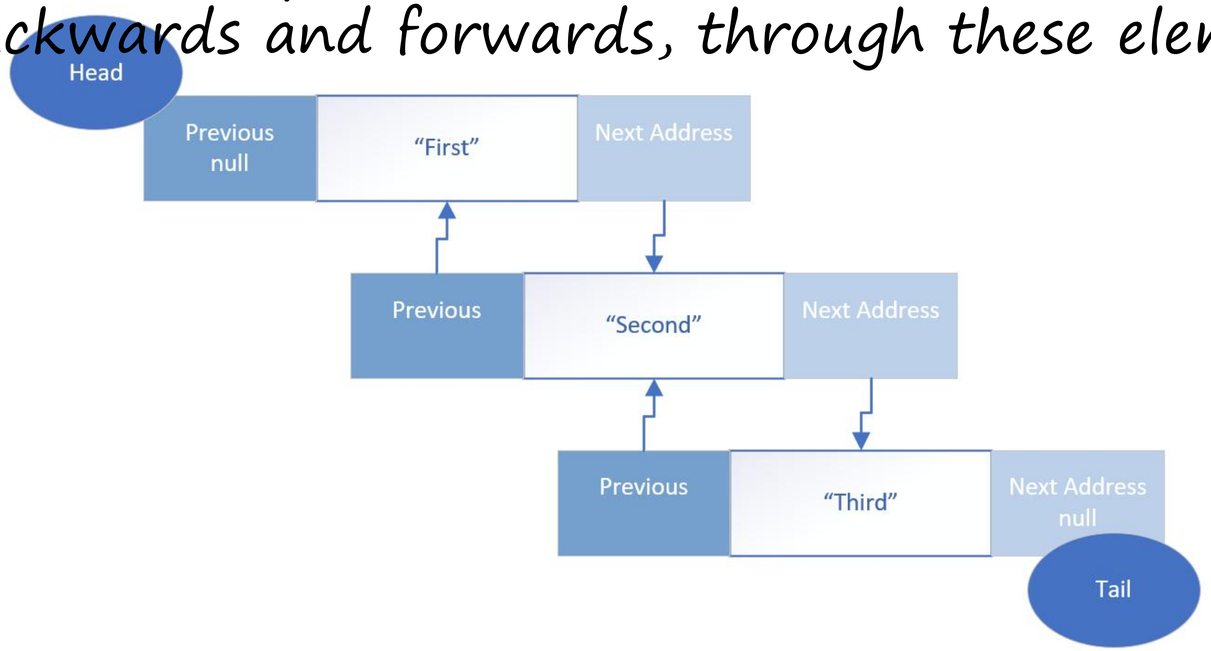


LinkedList

This architecture is called a doubly linked list, meaning an element is linked to the next element, but it's also linked to a previous element, in this chain of elements.

The beginning of the chain is called the head of the list, and the end is called the tail.

This can also be considered a queue, in this case, a double ended queue, because we can traverse both backwards and forwards, through these elements.



LinkedList - Retrieval of an Element costs more than an ArrayList r

Getting an element from the list or setting a value of element, isn't just simple math anymore with the LinkedList type.

To find an element, you'd need to start at the head or tail, and check if the element matches or keep track of the number of elements traversed, if we are matching by an index because the index isn't stored as part of the list.

For example, even if you know you want to find the 5th element, you'd still have to traverse the chain this way to get that fifth element.

This type of retrieval is considered expensive in computer currency, which is processing time and memory usage.

On the other hand, inserting and removing an element, is much simpler for this type of collection.

LinkedList - Inserting or Removing an Element may be less costly than using an Ar

In contrast to an ArrayList, inserting or removing an item in a LinkedList is just a matter of breaking two links in the chain, and re-establishing two different links.

No new array needs to be created, and elements don't need to be shifted into different positions.

A reallocation of memory to accommodate all existing elements is never required.

For a LinkedList, inserting and removing elements is generally considered **cheap** in computer currency, compared to doing these functions in an ArrayList.

LinkedList and ArrayList Operations - Big O

This slide shows the Big O values for the most common shared List operations or methods for both types.

For a LinkedList, adding elements to the start or end of the List will almost always be more efficient than an ArrayList.

	Linked List		ArrayList	
Operation	Worst Case	Best Case	Worst Case	Best Case
add()	O(1)		O(1)*	
add(int index, E element)	O(n)	O(1)	O(n)	O(1)*
contains(E element)	O(n)	O(1)	O(n)	O(1)
get(int index)	O(n)	O(1)	O(1)	
indexOf(E Element)	O(n)	O(1)	O(n)	O(1)
remove(int index)	O(n)	O(1)	O(n)	O(1)
remove(E element)	O(n)	O(1)	O(n)	
set(int index, E element)	O(n)	O(1)	O(1)	

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

LinkedList and ArrayList Operations - Big O

When removing elements, a LinkedList will be more efficient because it doesn't require re-indexing, but the element still needs to be found using the traversal mechanism, which is why it is O(n), as the worst case.

Removing elements from the start or end of the List will be more efficient for a LinkedList.

	Linked List		ArrayList	
Operation	Worst Case	Best Case	Worst Case	Best Case
add()	O(1)		O(1)*	
add(int index, E element)	O(n)	O(1)	O(n)	O(1)*
contains(E element)	O(n)	O(1)	O(n)	O(1)
get(int index)	O(n)	O(1)	O(1)	
indexOf(E Element)	O(n)	O(1)	O(n)	O(1)
remove(int index)	O(n)	O(1)	O(n)	O(1)
remove(E element)	O(n)	O(1)	O(n)	
set(int index, E element)	O(n)	O(1)	O(1)	

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

Things to Remember when considering whether to use an ArrayList vs Linke

The ArrayList is usually the better default choice for a List, especially if the List is used predominantly for storing and reading data.

If you know the maximum number of possible items, then it's probably better to use an ArrayList, but set it's capacity.

This code demonstrates how to set the capacity of your ArrayList to 500,000.

```
int capacity = 500_000;
ArrayList<String> stringArray = new ArrayList<>(capacity);
```

Things to Remember when considering whether to use an ArrayList vs Linke

An ArrayList's index is an int type, so an ArrayList's capacity is limited to the maximum number of elements an int can hold, Integer.MAX_VALUE = 2,147,483,647.

You may want to consider using a LinkedList if you're adding and processing or manipulating a large amount of elements, and the maximum elements isn't known but may be great, or if your number of elements may exceed Integer.MAX_VALUE.

A LinkedList can be more efficient when items are being processed predominantly from either the head or tail of the list

```
int capacity = 500_000;
ArrayList<String> stringArray = new ArrayList<>(capacity);
```

LinkedList

I've just talked in detail about how the LinkedList, and the ArrayList, are different under the covers.

An ArrayList is implemented on top of an array, but a LinkedList is a doubly linked list.

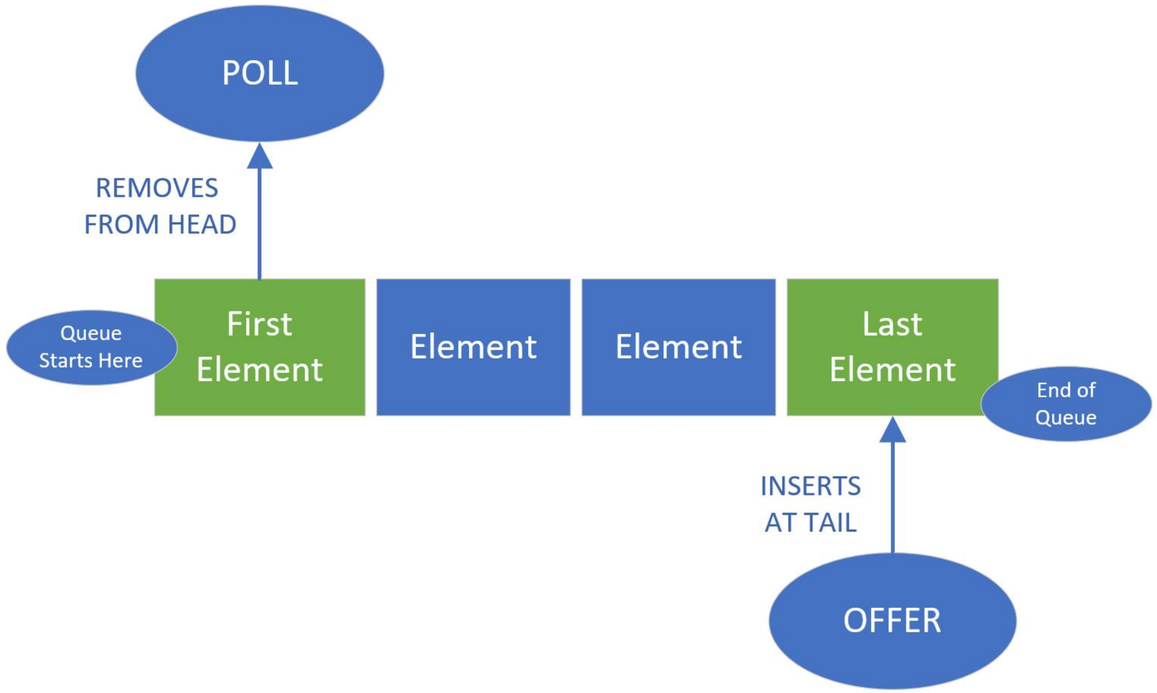
Both implement all of List's methods, but the LinkedList also implements the Queue and Stack methods as well.

A Queue is a First-In, First-Out (FIFO) Data Collec

When you think of a queue, you might think of standing in line.

When you get in a line or a queue, you expect that you'll be processed, in relationship to the first person in line.

We call this a First-in First-out, or FIFO data collection.

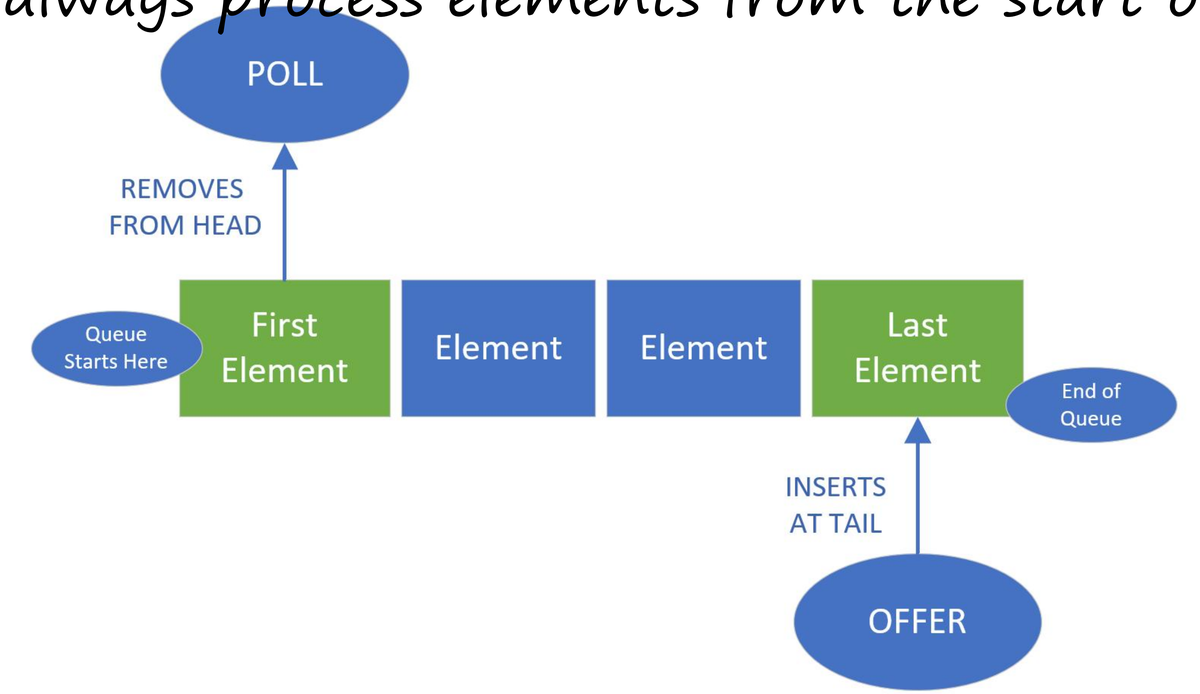


A Queue is a First-In, First-Out (FIFO) Data Collect

If you want to remove an item, you poll the queue, getting the first element or person in the line.

If you want to add an item, you offer it onto the queue, sending it to the back of the line.

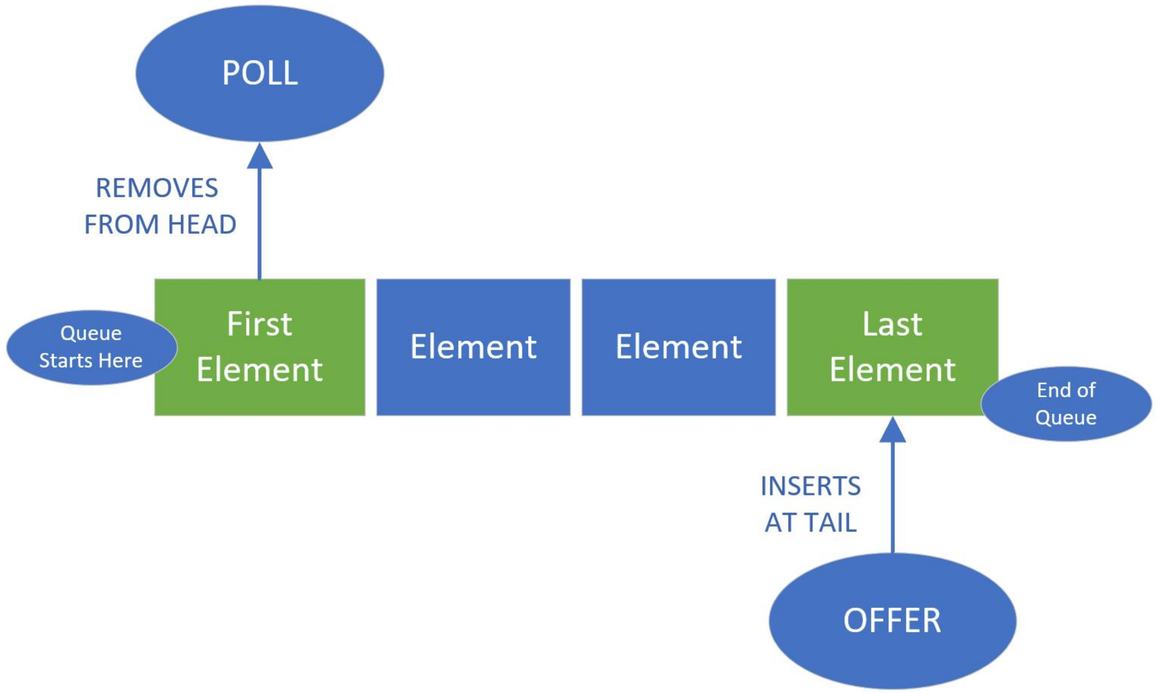
Single-ended queues always process elements from the start of the queue.



A Queue is a First-In, First-Out (FIFO) Data Collect

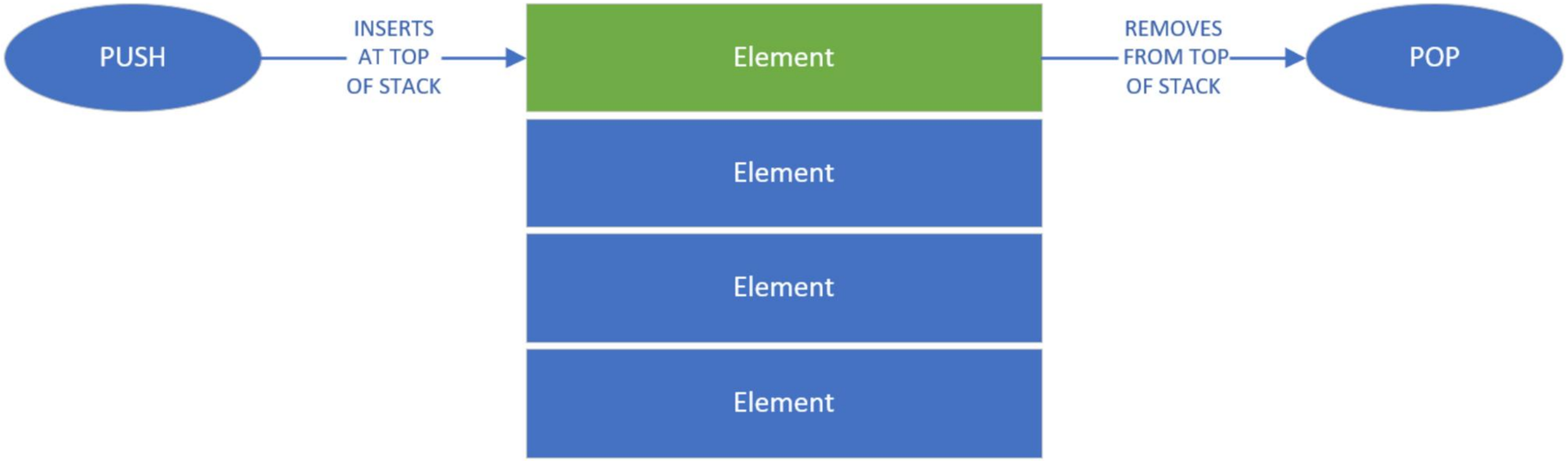
A double-ended queue allows access to both the start and end of the queue.

A LinkedList can be used as a double ended queue.



A Stack a is Last-In, First-Out (LIFO) Data Collect

When you think of a stack, you can think of a vertical pile of elements, one on top of another, as we show on this slide.



When you add an item, you push it onto the stack.

If you want to get an item, you'll take the top item, or pop it from the stack.

We call this a Last-In First-out, or LIFO data collection.

A LinkedList can be used as a stack as well.

What's an Iterator?

In a nutshell it's another way to traverse lists.

So far, we've mainly used for loops to traverse through elements in an array or list.

There is the traditional for loop and an index to index into a list.

And also, the enhanced for loop and a collection to step through the elements one at a time.

We've used both quite a few times in the course.

Let's look at how the Iterator works.

How does an Iterator work?

If you're familiar with databases, you might be familiar with a database cursor, which is a mechanism that enables traversal over records in a database.

An iterator can be thought of as something similar to a database cursor.

Specifically, an iterator is an object that allows traversal over records in a collection.

How does an Iterator work?

The Iterator is pretty simple.

When you get an instance of an iterator, you can call the **next** method to get the next element in the list.

You can use the **hasNext** method to check if any elements remain to be processed.

In the code, you can see a while loop which uses the iterator's **hasNext** method to determine if it should continue looping

In the loop, the **next** method is called and its value assigned to a local variable and the local variable printed out.

This prints each element in a list but does it through or via the iterator object.

How does an Iterator work?

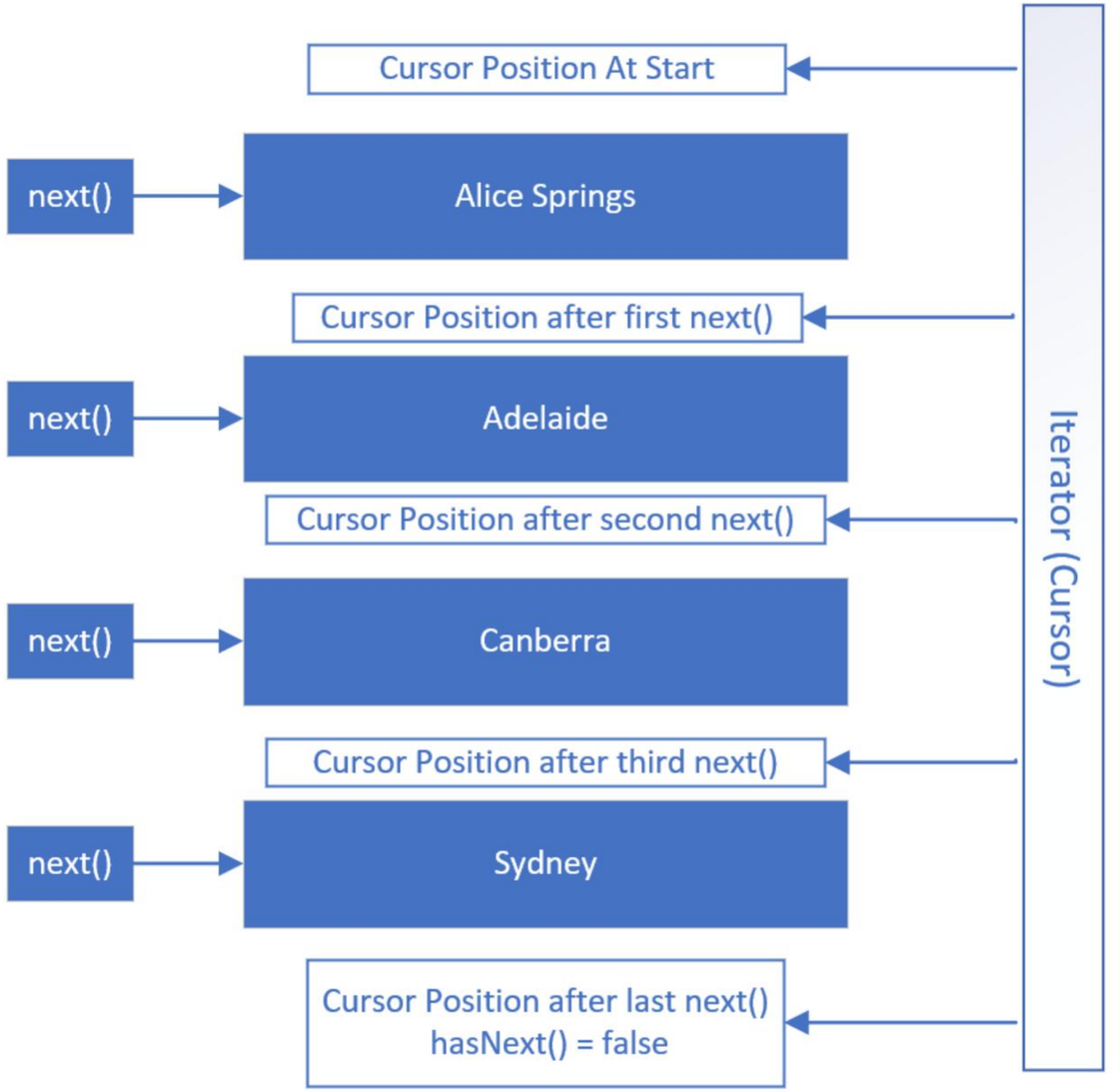
This slide shows visually how an Iterator works using the PlacesToVisit List.

When an iterator is first created, its cursor position is pointed at a position **before** the first element.

The first call to the **next** method retrieves the first element and moves the cursor position to be between the first and second elements.

Subsequent calls to the **next** method moves the iterator's position through the list, as shown, until there are **no elements left**, meaning **hasNext = false**.

At this point, the iterator or cursor position is below the last element.



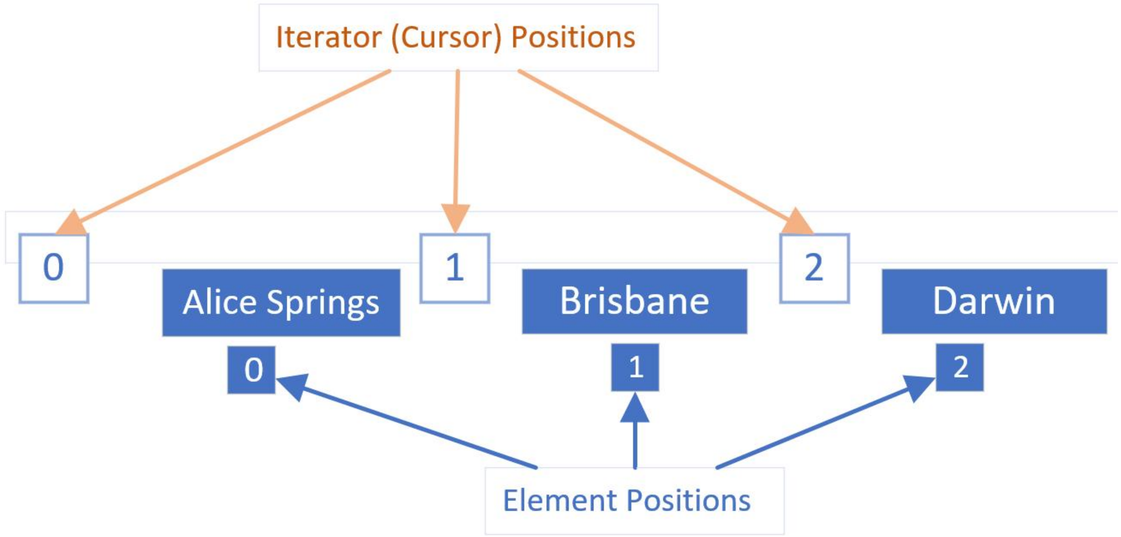
Iterator vs. ListIterator

An Iterator is forwards only and only supports the **remove** method.

A ListIterator allows you to navigate both forwards and backwards. Besides the **remove** method, it also supports the **add** and **set** methods, which function as you probably expect.

Iterator positions vs. Element positions

It's really important to understand that the iterator's cursor positions are **between** the elements.




```
var iterator = list.listIterator();

String first = iterator.next();  // Alice Springs returned, cursor moved to
// cursor position 1
String second = iterator.next(); // Brisbane returned, cursor moved to cursor
// position 2

// Reversing Directions
String reversed = iterator.previous(); // Brisbane returned, cursor moved to
// cursor position 1
```

LinkedList Challenge

It's now time for a LinkedList challenge.

I'm going to ask you to use LinkedList functionality to create a list of places, ordered by distance from the starting point.

I want you to use a ListIterator to move both backwards and forwards through this ordered itinerary of places.

LinkedList Challenge

First, create a type that has a field for a town or place name and a field for storing the distance from the start.

Next, create an itinerary of places or towns to visit, much like I've been doing in the last few videos.

Town	Distance from Sydney (in km)
Adelaide	1374
Alice Springs	2771
Brisbane	917
Darwin	3972
Melbourne	877
Perth	3923

LinkedList Challenge

But this time, instead of Strings, you'll want to create a LinkedList of your place or town type.

Here, I show a list of a few places in Australia and their distances from Sydney.

Town	Distance from Sydney (in km)
Adelaide	1374
Alice Springs	2771
Brisbane	917
Darwin	3972
Melbourne	877
Perth	3923

LinkedList Challenge

You'll create a LinkedList ordered by the distance from the starting point, in this case, Sydney.

Sydney should be the first element in your list.

You don't want to allow duplicate places to be in your list for this data set.

Town	Distance from Sydney (in km)
Adelaide	1374
Alice Springs	2771
Brisbane	917
Darwin	3972
Melbourne	877
Perth	3923

LinkedList Challenge

In addition, you'll create an interactive program with the following menu item actions:

```
Available actions (select word or letter):
(F)orward
(B)ackward
(L)ist Places
(M)enu
(Q)uit
```

You'll want to use a Scanner and the nextLine method to get input from the console.

You'll use a ListIterator to move forwards and backwards through the list of places on your itinerary.

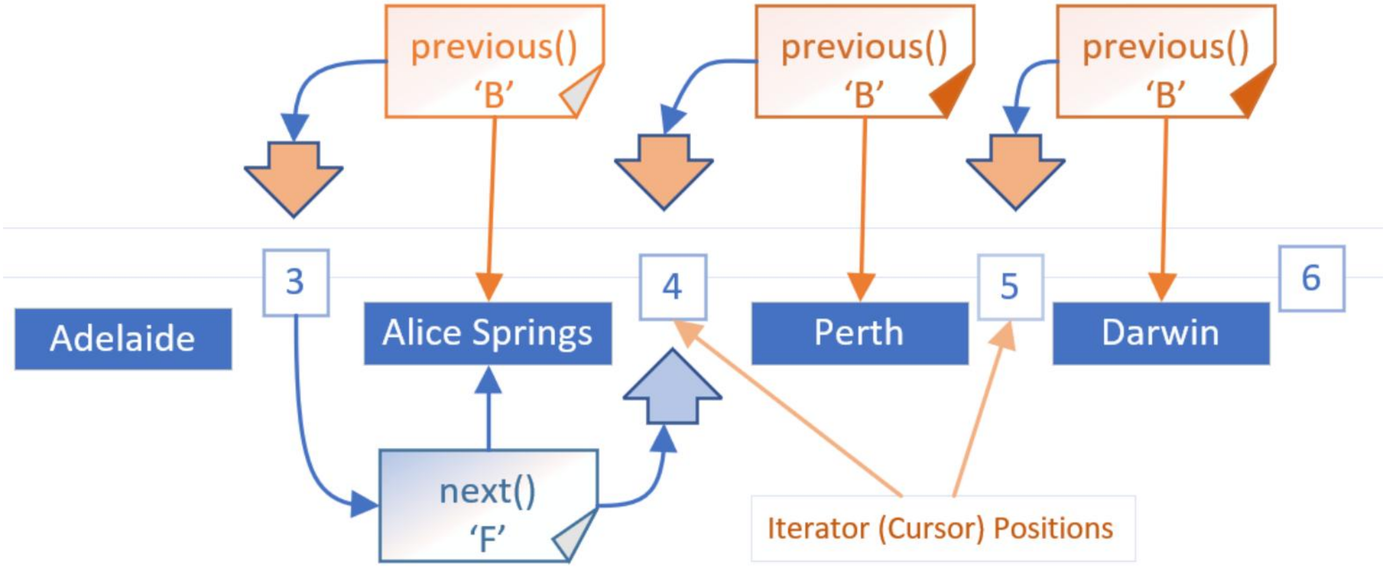
LinkedList Challenge

Now, I'll create the interactive part of our program, with the menu items shown

```
Available actions (select word or letter):
(F)orward
(B)ackward
(L)ist Places
(M)enu
(Q)uit
```

I'll use a Scanner to get input from the user and use a ListIterator to move back and forth through the itinerary.

Reversing Directions in a List Iterator

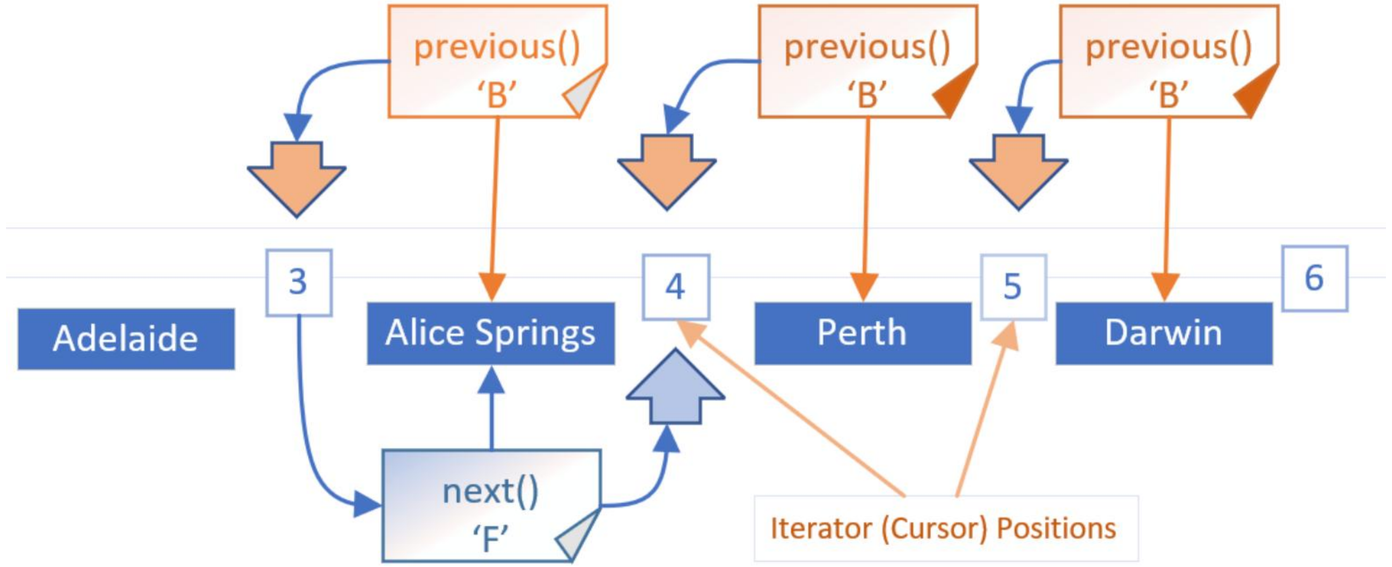


Here, I show the three B or back menu items, each calling the previous method.

This leaves our cursor position, shown as an orange arrow, positioned between Adelaide and Alice Springs.

When we decided to go forward, we called the next method from this cursor position and that gave us Alice Springs.

Reversing Directions in a List Iterator



When we change directions, we have to compensate for the cursor being ahead of where we want it to be.

So, we have to make the call, either next or previous, twice. The first time to adjust the cursor, and the second time to get the element.

Why does Java have primitive data types?

Some object-oriented languages don't support any primitive data types at all, meaning everything is an object.

But most of the more popular object-oriented languages of the day, including Java, support both primitive types and objects.

Primitive types generally represent the way data is stored on an operating system.

Primitives have some advantages over objects, especially as the number of elements you need to store increase.

Objects take up additional memory and may require a little more processing power.

We know we can create objects, with primitive data types as field types, for example, and we can also return primitive types from methods.

Why don't all of Java's collection types support primiti

When we look at classes like the ArrayList or the LinkedList, which we've reviewed in detail in this section, we find that these classes don't support primitive data types as a collection type.

In other words, **we can't do something** like creating a LinkedList, using an int primitive type.

As an example the code below won't compile.

```
LinkedList<int> myIntegers = new LinkedList<>();
```

This means, we can't take advantage of the great functionality that Lists provide, with primitive values. At least not directly.

Why don't all of Java's collection types support primiti

```
LinkedList<int> myIntegers = new LinkedList<>();
```

More importantly, we can't easily use primitives in some of the features we'll be learning about in the future, like generics.

But Java, as you have learned, does give us wrapper classes for each primitive type.

And we can go from a primitive to a wrapper, which is called boxing, or a wrapper to a primitive, which is called unboxing, with relative ease in Java.

What is Boxing?

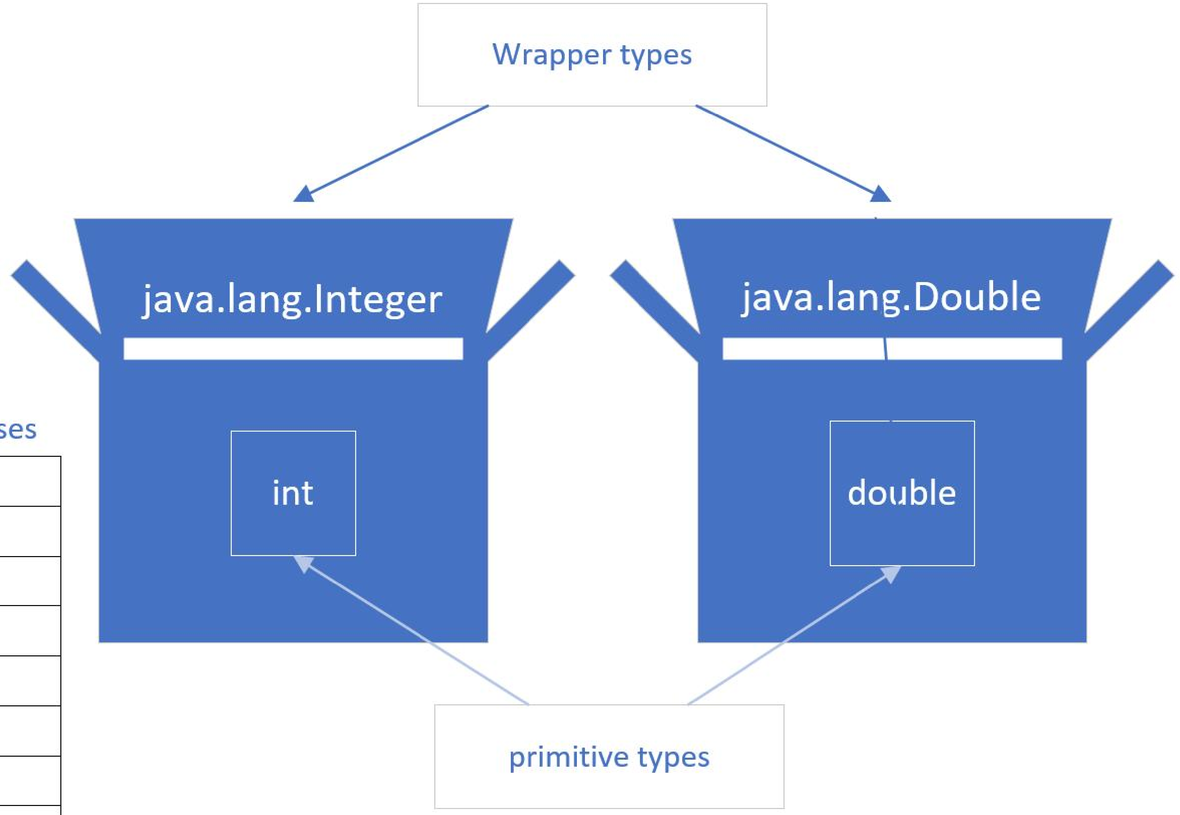
A primitive is boxed, or wrapped, in a containing class, whose main data is the primitive value.

Each primitive data type has a wrapper class, as shown on the list, which we've seen before.

Each wrapper type boxes a specific primitive value.

Wrapper Classes

Boolean
Byte
Character
Double
Float
Integer
Long
Short



How do we box?

Each wrapper has a static overloaded factory method, valueOf, which takes a primitive as an argument and returns an instance of the wrapper class.

The code shown on this slide, returns an instance of the java.lang.Integer class, to the boxedInt variable, with the value 15.

We can say the code below **manually boxes** a primitive integer.

```
Integer boxedInt = Integer.valueOf(15);
```

Deprecated Boxing using the wrapper constructor

Another manual way of boxing, which you'll see in older code, is by creating a new instance of the wrapper class, using the new keyword, and passing the primitive value to the constructor.

See this example below.

```
Integer boxedInt =  new Integer(15);
```

If you try this in IntelliJ, with any Java version greater than JDK-9, IntelliJ will tell you that this is deprecated code. And rightly so.

Deprecated Code

Deprecated code means it's outdated code and is likely to not be supported in a future version. It's been marked for deletion from the language at some point in the future.

If you come across deprecated code, there is usually always a newer, better way to do what you are trying to achieve, and you should use the new way.

Using new (with a constructor) is deprecated for wrapp

```
Integer boxedInt =  new Integer(15);
```

Java's own documentation states the following about the code above:

- It is rarely appropriate to use this constructor.
- The static factory valueOf(int) is generally a better choice, as it is **likely to yield significantly better space and time performance**.

This deprecation applies to all the constructors of the wrapper classes, not just the Integer class.

In truth, we rarely have to manually box primitives, because Java supports something called **autoboxing**.

What is autoboxing?

Autoboxing is where Java automatically boxes a primitive type for you. Hence the term autoboxing.

Java makes it easy to assign a primitive to a wrapper variable, as shown below.

```
Integer boxedInt = 15;
```

Java supports this syntax, and its actually preferred and in my opinion easier to read as well.

Underneath the covers, Java is doing the boxing. In other words, an instance of Integer is created, and its value is set to 15.

Allowing Java to autobox is preferred to any other method, because Java will provide the best mechanism to do it.

What is autoboxing?

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

Every wrapper class supports a method to return the primitive value it contains.

This is called unboxing.

In the example on this slide, I've autoboxed the integer value 15, to a variable called boxedInteger.

This gives us an object which is an Integer wrapper class, with the value of 15.

To unbox this on an Integer class, you can use the intValue method to do it manually, which returns the boxed value, the primitive int in this case.

What is autoboxing?

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

Just like boxing, it's unnecessary to **manually unbox**.

Automatic unboxing

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger;
```

Automatic unboxing is really just referred to as unboxing in most cases.

You can assign an instance of a wrapper class directly, to a primitive variable.

The code on this slide shows an example.

We're assigning an object instance to a primitive variable, in the second statement.

Automatic unboxing

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger;
```

This is allowed, because the object instance is an Integer wrapper, and we're assigning it to an int primitive type variable.

Again, this is the preferred way to unbox a wrapper instance.

Let's get back to some code now and see different examples of autoboxing and unboxing in action.

Autoboxing Challenge with ArrayLists

Okay, so it's time for a challenge on autoboxing and unboxing.

In this challenge, you'll need to create a simple banking application with a Customer and Bank type.

The **Customer** will have a name and an **ArrayList** of transactions containing **Double** wrapper elements.

- A customer's transaction can be a credit, which means a positive amount, or it can be a debit, a negative amount.

Autoboxing Challenge with ArrayLists

The **Bank** will have a name, and an **ArrayList** of customers.

- The bank should **add a new customer**, if they're not yet already in the list.
- The bank class should allow a customer to **add a transaction**, to an existing Customer.
- This class should also **print a statement**, that includes the customer name, and the transaction amounts. This method should use unboxing.

Enumeration

The enum type is Java's type to support something called an enumeration.

Wikipedia defines enumeration as, *"A complete ordered listing of **all the items** in a collection."*

The enum type

Java describes the enum type as: A special data type that contains predefined constants.

A constant is a variable whose value can't be changed, once its value has been set.

An enum is a little like an array, except its elements are known, not changeable, and each element can be referred to by a constant name, instead of an index position.

The enum type

```
public enum DayOfTheWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

An enum in its simplest form, is described like a class. However, the keyword enum replaces the keyword class.

You can name the enum with any valid identifier, but like a class, upper camel case is the preferred style.

Within the enum body, you declare a list of constant identifiers, separated by commas. By convention, these are all uppercase labels.

One example of an enum, is the days of the week, as shown here.

The enum type

```
public enum DayOfTheWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

An enum is ordered, by the way you declare the constants.

This means that SUNDAY is considered the first day of the week, and SATURDAY is the last day of the week, in this example.

The enum type

The enum type is used to declare a limited set of constants, and sometimes there is a natural order to the listing, as in the case of days of the week.

Some other examples of possible enum declarations are:

- The months in the year: JANUARY, FEBRUARY, MARCH, etc.
- The directions in a compass: EAST, NORTH, WEST, SOUTH.
- A set of sizes: EXTRA_SMALL, SMALL, MEDIUM, LARGE, EXTRA_LARGE.

The enum type

Underneath the covers, the enum type is a special type of class, which contains fields to support the constants. I'll get into that in a later discussion.

You don't have to understand all the internals of an enum, to derive the benefits of using the type.

Once you get used to how this type works, you will probably find many places to use an enum.

They simplify your code and make it more readable in many ways.