NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 7: 12th Mar 2024

Java Array vs Java List

An array is immutable, and we saw, that we could set or change values in the array, but we could not resize it.

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?

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.

The ArrayList

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

It keeps track of the capacity, which is the actual 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 all happens 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.

An array requires square brackets in the declaration.

In the new instance, square brackets are also required, with a size specified inside

An ArrayList should be declared, with the type of element in 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.

We 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
```

```
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
```

```
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.

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
```

```
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 Array: `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);`

Getting a String representation for Multi-Dimensional Arrays and ArrayLists

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);
```

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)`

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.

The ArrayList Challenge

The challenge is to create an interactive console program.

And give the user a menu of options as shown here:

```
Available actions:

0 - to shutdown

1 - to add item(s) to list (comma delimited list)

2 - to remove any items (comma delimited list)

Enter a number for which action you want to do:
```

Using the Scanner class, solicit a menu item, 0, 1 or 2, and process the item accordingly.

The ArrayList Challenge

Your grocery list should be an ArrayList.

You should use methods on the ArrayList, to add items, remove items, check if an item is already in the list, and print a sorted list.

You should print the list, sorted alphabetically, after each operation.

You shouldn't allow duplicate items in the list.

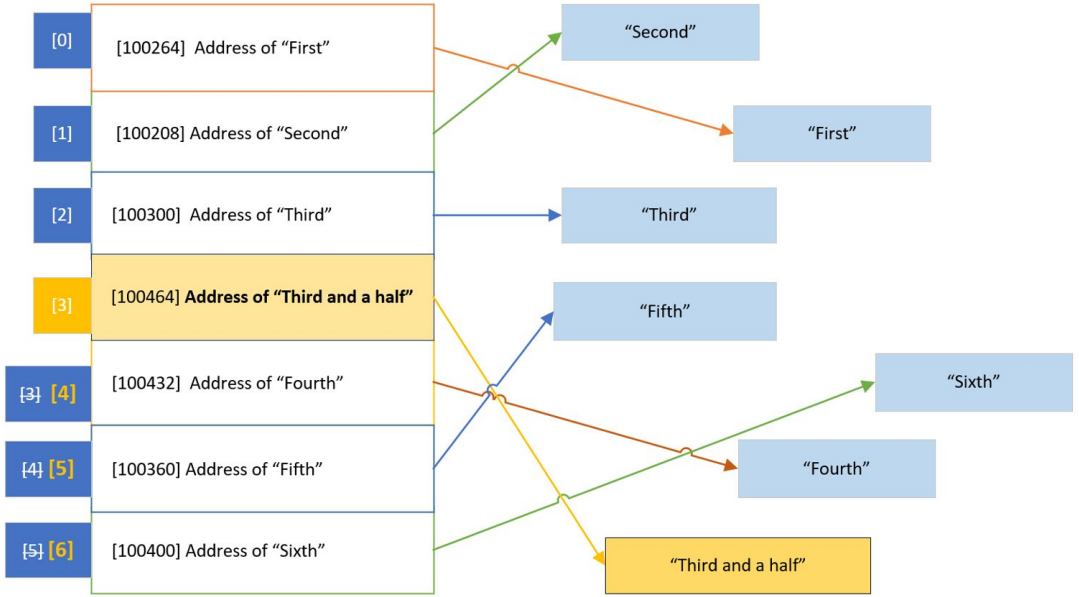
Array of primitive values

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

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.
- ArrayLists are really implemented with arrays, under the covers.



Arrays and ArrayLists of reference types

This is a cheap 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.

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 process, if the number of elements is large.

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 we're passing 10 in the constructor of this list.

We 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

We 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 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 re-allocates 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.

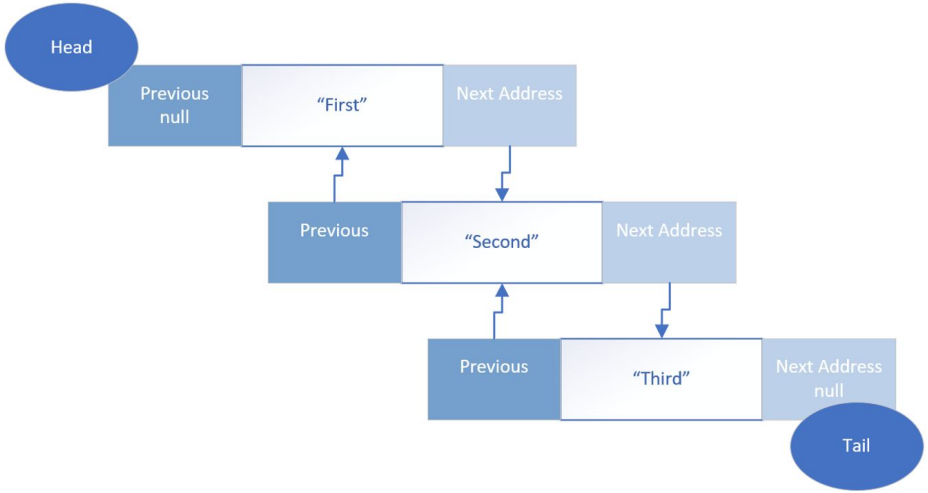
We can't actually get this capacity size, from the ArrayList.

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.

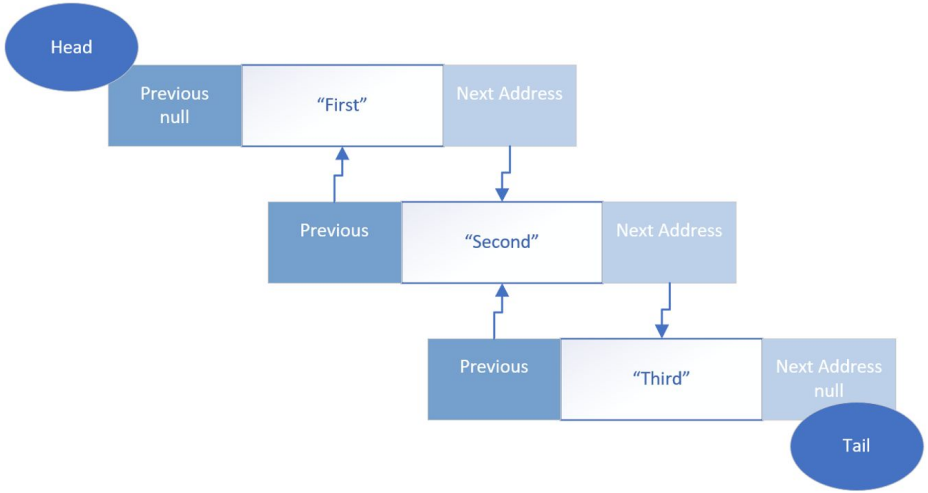


LinkedList

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 retrieval

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, we'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 ArrayList

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.

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

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

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.

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.

LinkedList

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

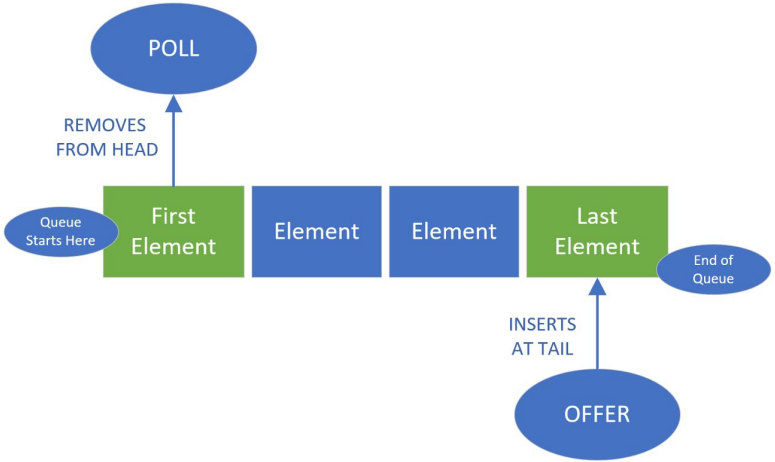
Both implement all of List's methods, but the LinkedList also implements the Queue and Stack methods as well.

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

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.

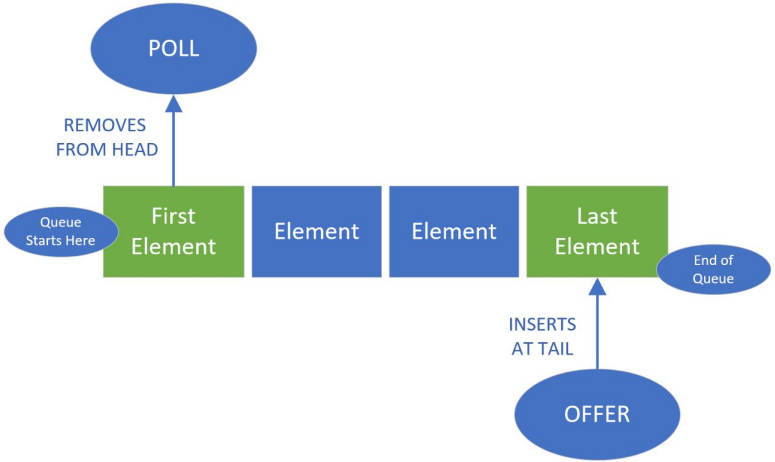


A Queue is a First-In, First-Out (FIFO) Data Collection

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.



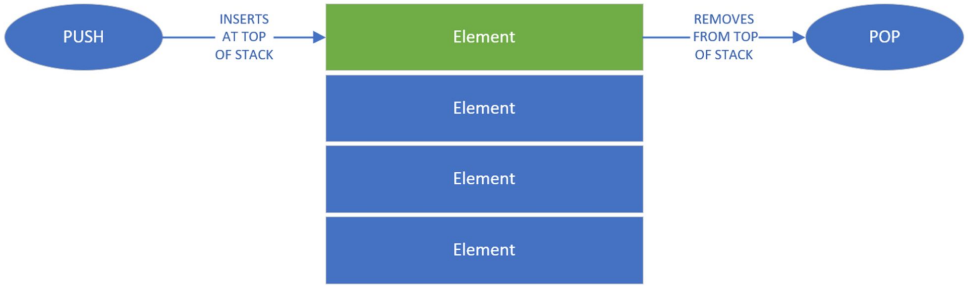
A Queue is a First-In, First-Out (FIFO) Data Collection

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 Collection

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?

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

We can use the traditional for loop and an index, to index into a list.

We can use the enhanced for loop and a collection, to step through the elements, one at a time.

But Java provides other means to traverse lists.

Two alternatives are the Iterator, and the ListIterator.

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 similar to a database cursor.

The kind of cursor we're referring to here, can be described as 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 would just print each element in a list, but do it through the iterator object.

How does an Iterator work?

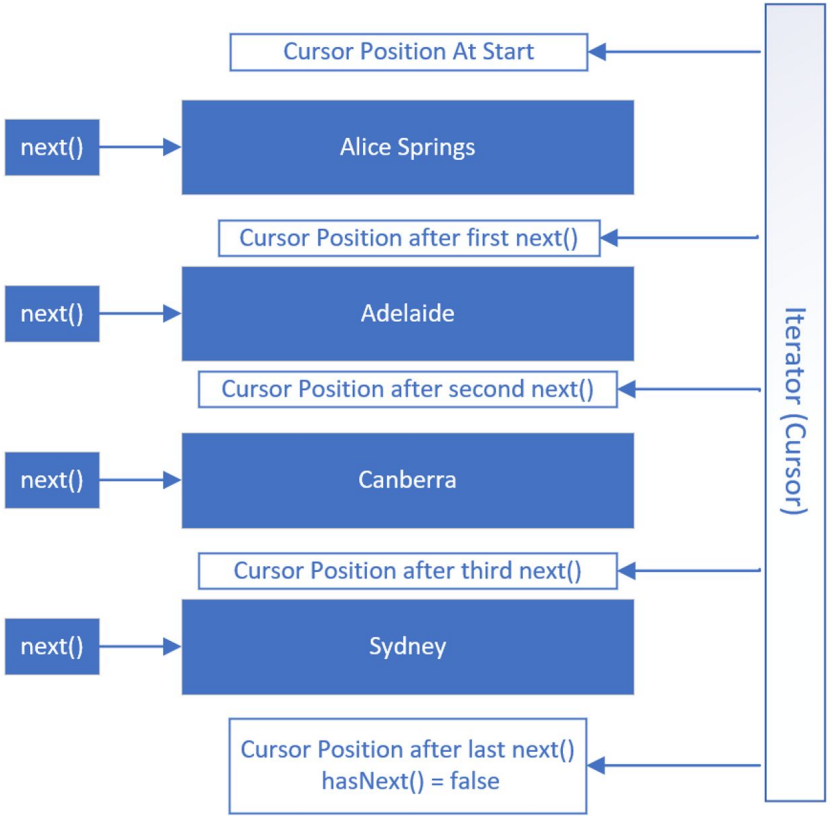
This slide shows visually how an Iterator works, using the PlacesToVisit List.

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

The first call to the **next** method gets 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.



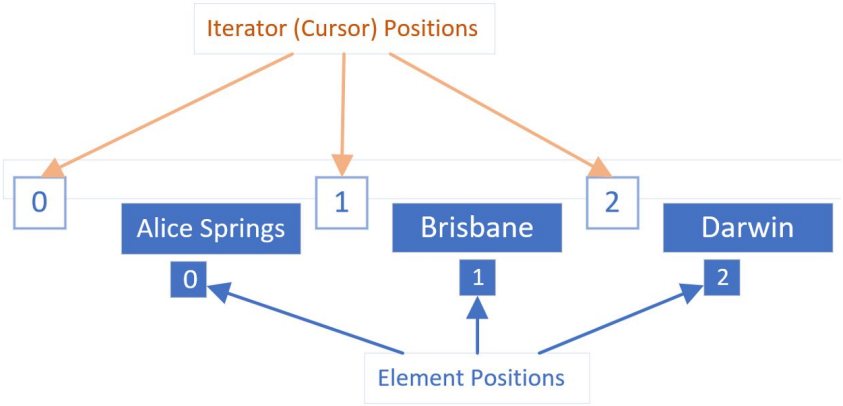
Iterator vs. ListIterator

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

A ListIterator can be used to go both forwards and backwards, and in addition to the **remove** method, it also supports the **add** and **set** methods.

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.

And we want 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 town or place name, and a field for storing the distance from the start.

Next, create an itinerary of places or towns to visit.

Create a LinkedList of your place or town type.

Here we 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.

LinkedList Challenge

In addition, you'll create an interactive program with the following menu item options.

```
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.

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, support both primitive types and objects, as does Java.

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

Primitives have some advantages over objects, especially as the magnitude, or number of elements increase.

Objects take up additional memory, and may require a bit 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 primitives?

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

In other words we can't do the following, creating a LinkedList, using the int primitive type.

This code won't compile.

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

This means, we can't use all the great functions Lists provide, with primitive values.

Why don't all of Java's collection types support primitives?

```
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 we know, gives us wrapper classes for each primitive type.

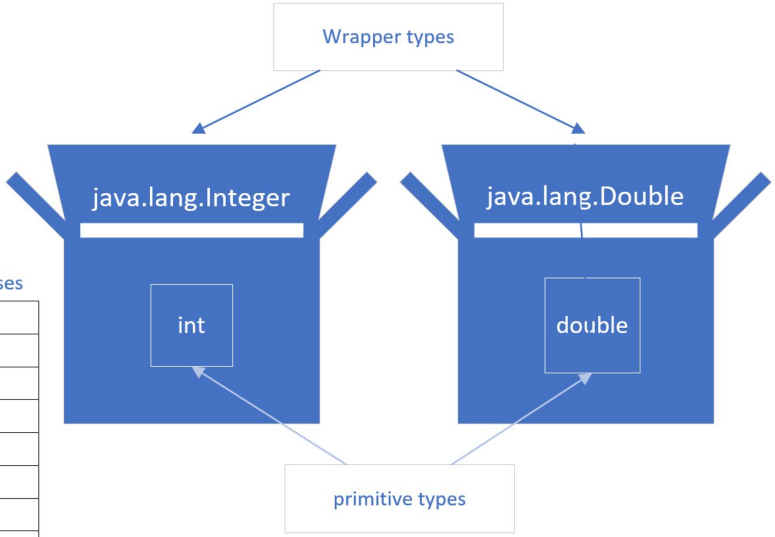
And we can go from a primitive to a wrapper, which is called boxing, or a wrapper to a primitive, which is called unboxing, with relative ease in Java.

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, 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 in it.

We can say this code **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.

We show an example of this here.

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

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

Deprecated Code

Deprecated code means it's older, and it may not be supported in a future version.

In other words, you should start looking for an alternate way of doing something, if it's been deprecated.

Using new (with a constructor) is deprecated for wrappers

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

Java's own documentation states the following:

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?

We can simply assign a primitive to a wrapper variable, as we show on this slide.

```
Integer boxedInt = 15;
```

Java allows this code, and it's actually preferred, to manually boxing.

Underneath the covers, Java is doing the boxing. In other words, an instance of Integer is created, and it's 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, we've autoboxed the integer value 15, to a variable called boxedInteger.

This gives us an object which is an Integer wrapper class, and has the valueof 15.

To unbox this, on an Integer class, we can use the intValue method, which returns the boxed value, the primitive int.

What is autoboxing?

```
Integer boxedInteger = 15;

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

This method is called **manually unboxing**.

And like boxing, it's unnecessary to manually unbox.

Automatic unboxing

```
Integer boxedInteger = 15;

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

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

We 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.intValue();
```

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.

Autoboxing Challenge with ArrayLists

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

In this challenge, you will 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 it's value has been assigned.

So an enum is a little like an array, except it's 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, but the keyword enum, replaces the keyword class.

You can name the enum with any valid identifier, but like a class, Upper CamelCase 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.

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 might be:

- 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, but we'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 this type.

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

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