# Collections

Now that we've learned about how to write clean code by leveraging the principles of object-oriented programming, it's time to start talking about data. After all, the programs that keep the world running aren't solely focused on taking simple inputs and using them once or twice. In reality, most apps that you interact with daily involve searching, sorting, and tracking down details across data sets that feature thousands of data points. Some of these data points are as simple as numbers on a spreadsheet, while others are complex objects that contain their own logic, methods, and attributes.

To keep our applications performant, we need to start thinking about how we structure the storage of our information and the methods that access it. Luckily, Java has provided a framework of **Collections** to help us get the job done. The two root interfaces of Java collection classes are the Collection interface (**java.util.Collection**) and the Map interface (**java.util.Map**).

## What are Collections?

A collection is an object that represents a group of objects. In Java, these collections have been gathered into the **collections framework**. Essentially, we use this **framework (aka, set of classes and interfaces)** to represent and manipulate collections of data independently of their implementations. In practice, this means working with larger data sets in flexible ways to get work done within our applications.

InfoWarningTip

If answering the above question seems too difficult, check out the Extra Resources at the bottom of this page for a refresher/good explanation.

You can start your understanding of collections as being like arrays, but with way more features and flexibility. Before we get into the details, here are some of the advantages of using collections in our programs:

* Collections reduce the amount of programming we have to do by **implementing data structures and algorithms, so we don't have to write them ourselves**.
* They increase **performance** by providing their own tried-and-tested implementations of data structures and algorithms, allowing for **interchangeable implementations across each interface**. This means we can choose the best tools for the job, again, without having to write them ourselves.
* Collections **provide a common language to pass information between unrelated** [**APIs**](https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces) (Application Programming Interfaces). This also makes it easier to learn how to use APIs because we can learn one set of skills and apply them across various collections.
* Collections also aid in abstraction, as they **give us a standard interface** for collections and the algorithms with which we manipulate them.
* Collections work to **overcome the limitations of basic arrays** such as their fixed indices.

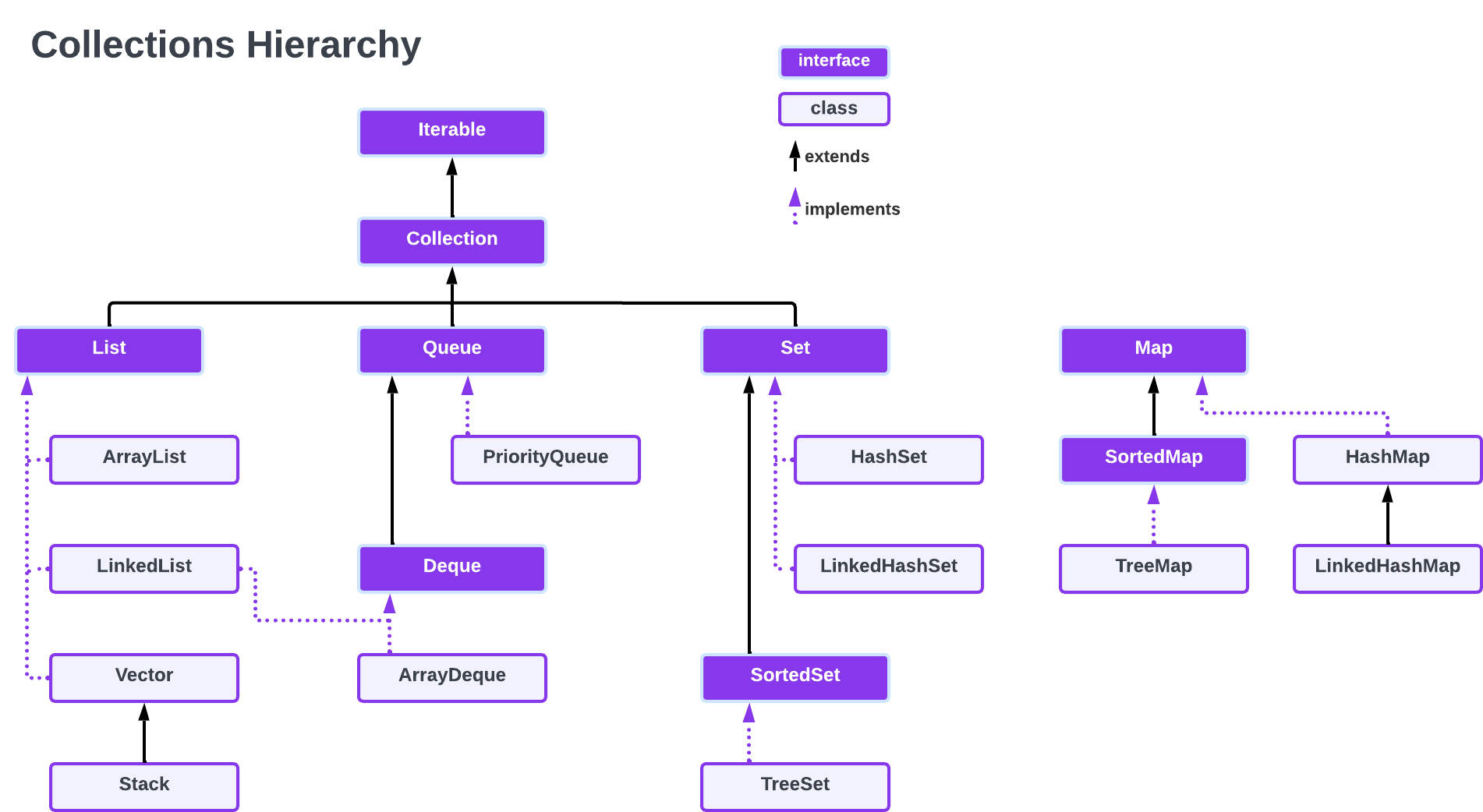
## The Java Collections Framework

The collections framework is made up of the following pieces:

* **Collection Interfaces**. These represent the various types of collections, including sets, lists, and maps. These are the basis for the rest of the framework.
* **General-purpose Implementations**. These are the primary implementations of the collection interfaces–how we use them.
* **Algorithms**. These are static methods that help us with standard tasks like searching and sorting through lists of objects.
* **Array Utilities**. These provide collections-like functions for arrays of primitive types and reference objects. While this isn't technically a part of the collections framework, it usually gets listed alongside the above because of how they rely on some of the same infrastructure.

## The Collections Hierarchy

We'll be reviewing most of the collections hierarchy throughout this week. Here's what it looks like all laid out in terms of classes and interfaces:



Here are the basics about each of the primary interfaces:

* **List:** A list is an **ordered** collection of objects that **allows for duplicate values**. It preserves the insertion order (i.e., the order in which the objects were inserted into the list), which means we can search it using positional access.
* **Queue:** A queue holds the elements about to be processed. Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the head of the queue is that element that would be removed. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.
* **Set:** A set is an **unordered** collection of objects that **does not allow for duplicate values**. Its strengths lie in that simplicity.
* **Map:** Map is more of a function/method than a collection, but it goes along with collections well enough that it makes sense to teach it now. The map interface represents a direct mapping from one key to one value. As a result, a map cannot contain duplicate keys. The map interface includes methods for operations such as put, get, and remove and even has some bulk operations and methods for viewing collections. Essentially, a map will help us work with the above collection interfaces to leverage their unique properties.

# **ArrayList Basics**

This lesson will generally describe the List Interface, but it's often better to learn about these abstractions using an example. Therefore, we're going to start off with the basics of ArrayList. Hopefully, our more general description of the List interface will make more sense once you've internalized some of the specifics that refer to the ArrayList, which is an implementation of the more general purpose List interface.

An ****ArrayList**** (also referred to as a list) is a resizable array where the resizing is performed behind the scenes. The underlying array grows or shrinks as needed, and elements shift automatically to create or remove gaps. You can use an ArrayList instead of an Array at any time, but ArrayLists are especially useful ****when you don’t know the size of the array you need or when you need to add and remove items frequently.****

Like arrays, ArrayList elements are indexed, beginning at 0. When inserting or removing items from a list, index values of elements will change frequently because the items will shift to create or remove gaps. As a result, the size of an ArrayList also fluctuates.

ArrayLists are declared and instantiated as follows:

CC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

1

ArrayList<E> listName = new ArrayList<E>();

In the above, E represents an ****object type****, like String, Integer, or a class of our own creation. The size of the list, when first initialized, is 0. ****Note: primitives do not work with**** ArrayLists. If you want to put primitives in an ArrayList, you must wrap the values to become ****Objects forms, such as**** Integer ****for**** int.

Once instantiated, ArrayLists call methods with the dot operator to access, modify, or remove items in the list. The general syntax of an ArrayList method call is listName.methodName(args…)

The following ArrayList methods are often used, so we've demonstrated them in the code samples below. Access the full [Java Quick Reference](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) to view the full list of methods if you like. Note: In the examples below, E represents whatever data type is being stored in the ArrayList:

* int size()****:**** returns the number of elements in the list.
* boolean add(E obj)****:**** appends obj to the end of the list and returns true if successful.
* void add(int index, E obj)****:**** moves any current objects at the given index or beyond to the right (i.e., to a higher index) and inserts the given obj at the given index.
* E remove(int index)****:**** removes the item at the index and shifts the remaining items to the left (i.e., to a lower index).
* E get(int index)****:**** returns the item in the list at the given index.
* E set(int index, E obj)****:**** replaces the item at the given index with the obj you've passed in as a parameter.

Run the code as is, then play around with the specific parameters to see the effects. Can you cause some errors? What did you do, and how do you fix them? You're going to be working with Collections rather extensively from here on out, so get in there and get your hands dirty:

import java.util.\*; // This is the required import for an ArryaList and/or List

public class ArrayListBasics {

public static void main(String[] args) {

// DECLARE, INITIALIZE, SIZE, and PRINT

ArrayList<String> nameList = null; /\* Declare \*/

System.out.println(nameList);

nameList = new ArrayList<String>(); /\* Allocate memory\*/

System.out.println(nameList);

System.out.println("size is " + nameList.size());

// ADD w/o index

nameList.add("Andrea");

System.out.println(nameList);

nameList.add("Bob");

System.out.println(nameList);

nameList.add("Carrie");

System.out.println(nameList);

nameList.add("Eduardo");

System.out.println(nameList);

System.out.println("size is " + nameList.size());

// ADD w/ index

nameList.add(3, "Dierdre");

System.out.println(nameList);

System.out.println("size is " + nameList.size());

// GET

System.out.println("name at index 2 is " + nameList.get(2));

// SET

nameList.set(1, "Tori");

System.out.println(nameList);

System.out.println("size is " + nameList.size());

// REMOVE

nameList.remove(3);

System.out.println(nameList);

System.out.println("size is " + nameList.size());

}

}

## Wrapper Classes

**ArrayLists can only hold object types, so they cannot be created using primitive types like** int**,** double**,** boolean**, etc**. Luckily, the Wrapper classes we've already learned about can be used to seamlessly store primitive values into lists. For example, the Integer class can be used to store int values and the Double class can be used to store double values, as shown in the code below.

import java.util.\*; // REQUIRED IMPORT for ArryaList and List

public class WrapperBasics {

public static void main(String[] args) {

// DECLARE, INITIALIZE, SIZE, and PRINT

ArrayList<Integer> numbers = null;

System.out.println(numbers);

numbers = new ArrayList<Integer>();

System.out.println(numbers);

System.out.println("size is " + numbers.size());

// ADD w/o index

numbers.add(1); // Autoboxing example

System.out.println(numbers);

numbers.add(3);

System.out.println(numbers);

numbers.add(4);

System.out.println(numbers);

numbers.add(Integer.valueOf(5)); // no boxing necessary

System.out.println(numbers);

System.out.println("size is " + numbers.size());

// ADD w/ index

numbers.add(1, 2);

System.out.println(numbers);

System.out.println("size is " + numbers.size());

// GET

int x = numbers.get(2); // Unboxing example

System.out.println("number at index 2 is " + x);

// SET

numbers.set(3, 100);

System.out.println(numbers);

System.out.println("size is " + numbers.size());

// REMOVE

numbers.remove(3);

System.out.println(numbers);

System.out.println("size is " + numbers.size());

}

}

## **List Interface**

Now that you've seen the ArrayList in action, it should be easier to understand its more abstract parent interface: the List.

As discussed in the previous lesson, ArrayList ****implements**** the List interface. That means that everything described above is one way to utilize the List interface: turning the List interface's super powers into an object that you can work with. By itself, the List interface represents an ****ordered**** collection of items that also keeps track of the order in which items were added to the given list. It ****extends**** the Collection interface, but, as an interface, cannot be used directly to create objects. Instead, we create objects using the following classes that implement List's functionality:

* ****ArrayList**** is a resizable array found in the java.util package, and behaves as described in this lesson. Technically speaking, when we use an ArrayList and add a new item. If no more available slots exist, a new array with more slots is created, and the old array is copied into the new one. Finally, the old array is left to be garbage collected later.
* ****LinkedList**** is very similar to the ArrayList and can be used in similar ways, but the way that it works behind the scenes is quite different. Rather than having an array inside of an object as the ArrayList does, the LinkedList works by stringing together a series of ****containers****. Each container contains a link to the next container in the list, so when we add a new element to the LinkedList, it simply creates a new container and links it to the previous element. This means in practice that people generally use ArrayList for ****data storage****, but they use LinkedList for ****data manipulation.**** We'll talk more about LinkedLists in the lessons that follow.
* ****Vector**** can be thought of as a precursor to ArrayList. Essentially, it is a less-optimized version of the ArrayList, so we won't really be going into much detail here. If you'd like to read more, here's a [short tutorial](https://www.programiz.com/java-programming/vector) about it.
* ****Stack**** represents a Last In, First Out (LIFO) List. That means when you push a new element to the list, it ends up at the top. The same goes for if you remove (pop) an item from the list–it'll pop right off of the top. This is an extension of the Vector class, so it isn't much used anymore. Here's a [short tutorial](https://www.programiz.com/java-programming/stack" \l ":~:text=The Stack class provides the,stack data structure in Java.) for those of you interested in it.

The List interface comes with its own built-in methods that are passed down to its subclasses:

* boolean add(ElementType element)****:**** This adds an element to the given list according to the rules that govern the given list type.
* boolean addAll(int index, Collection c)****:**** This adds a collection to the given index. You can use this one to add multiple values to a list you've already created.
* object remove(int index)****:**** This removes whatever object exists at the given index.
* object get(int index)****:**** This returns the value of the object at the given index.
* object set(int index, Obj obj)****:**** This overwrites the object at the given index with the object obj you provide.
* int indexOf(Object obj)****:**** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
* int lastIndexOf(Object obj)****:**** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

There are others, but these are the basics. We'll be talking about more implementations of the List interface as we continue throughout the week. Remember to watch the differences between the various lists/collections and think about when they will be helpful.

## **ArrayList or List?**

Technically, when declaring any list (ArrayList, LinkedList, etc.), you can do so by using the List interface directly or the specific interface that you're going to use. Here's an example of the two options for declaring an ArrayList:

CC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

1

2

List<T> variableName = new ArrayList<T>();

ArrayList<T> variableName = new ArrayList<T>();

So what's the difference? The difference is rooted in the behaviour of interfaces and polymorphism. If you declare an interface and initialize it with an implementing class, that class object would only have access to the methods that the List interface itself features. Because ArrayList is essentially a "child" of the List interface, implementing ArrayList will give you access to all of those "parent" methods, as well as everything specific to the ArrayList interface itself. Remember: just like child classes inherit all of the "super" powers of their parent/ancestor classes, so do the sub-interfaces inherit the methods of their "parents". When you implement an interface, think about what you'll need. If you need to be able to switch out implementations for the interface easily, then it's better to "declare" the object as an ****interface**** on the left-hand side and pass an implementing ****class**** on the right–as shown in the first option above. On the other hand, if you need specialized implementations for the interface's methods from the child class and won't care much about switching out implementing classes, then declare the object using the class type–as shown in option 2 above.

The LinkedList

ArrayList is a big improvement over regular Arrays in Java. It solves our issue of resizability and makes it so we can add or remove elements without having to worry about taking up unnecessary amounts of memory.

But it isn't perfect. One problem with Arrays is that they have to be stored all together, with their elements kept in the sequential order that we've defined for them. In practice, this is far more complicated than you might realize. For instance, when we insert an element into an Array, it essentially shifts all the elements to the right to make room to insert the new element into the indicated slot. Sometimes it even creates a whole new array and deletes the old one when there isn't enough room in the current reference.

Indeed, with ArrayList, such an array is being created behind the scenes every time we make a change to our ArrayList variable, whether it's taking something out or adding something new in. Again, this might not seem like such a big deal when your arrays only contain a few integers, but imagine them on a grander scale, with thousands of entries and you start to see the problem. In the end, the larger the array, the more data is being copied, pasted, and unnecessarily hogging the CPU power we need to efficiently run our programs.

This is why the LinkedList exists: to make manipulation of data in large lists possible without requiring so much recopying.

InfoWarningTip

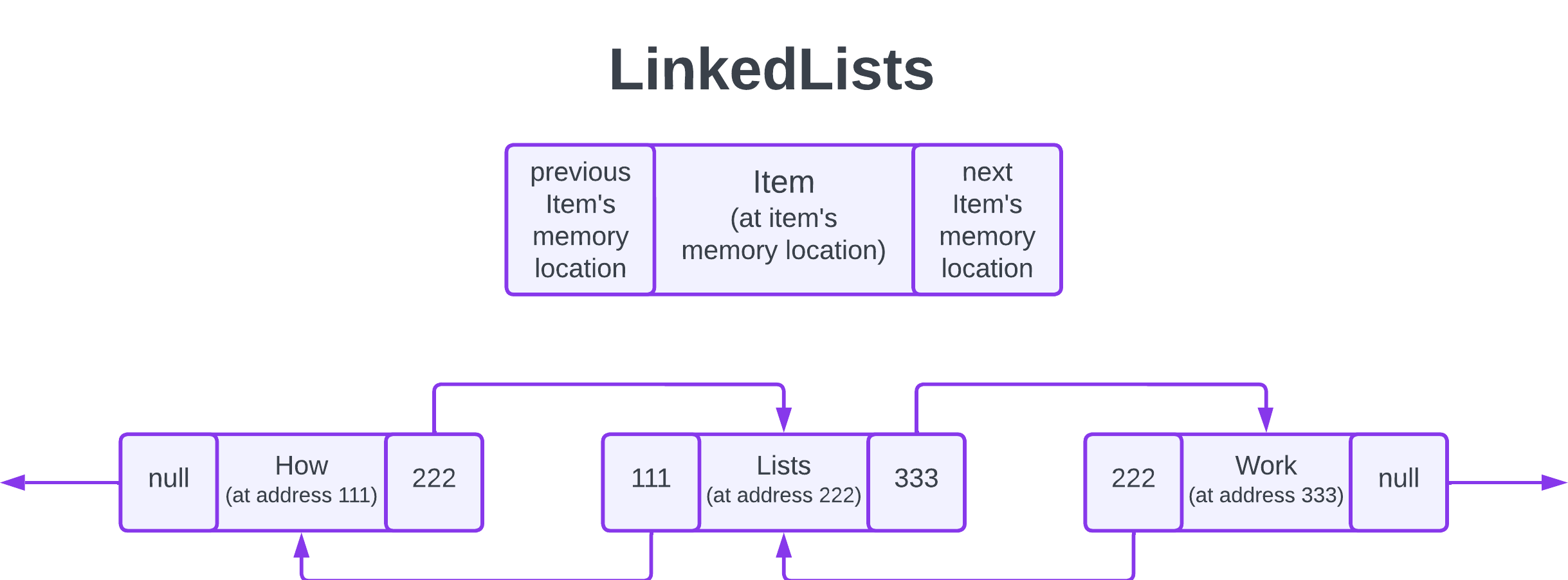
As we progress through the rest of the Java Collections, you might want to keep a reference [like this one](https://www.codejava.net/java-core/collections/java-collections-framework-summary-table) nearby. Each collection has its own optimizations and use-cases. You should watch out for them, and describe them to yourself in words you understand. For example, "LinkedLists are generally easier on memory than ArrayLists when items are being added or removed frequently," might be a good start for this lesson.

## LinkedList and Memory

While an Array stores its data as one unit at one position in your computer's memory, a LinkedList stores its information by linking each data point in the list to at least one other data point in the same list. In this way, they're not bound to each other in one place in memory. While we could spend a few dozen paragraphs explaining the ins and outs of the LinkedList, it would probably be better to watch someone work with the two directly. Follow along with the examples in John's video here, then continue reading below:

To recap:

* While there are various implementations of linked lists throughout computer science, Java's implementation is what's known as a **doubly linked** list: every element in the LinkedList knows where the elements that come **directly before** and **directly after** it live in memory. Note: the first and last elements have null pointers for the nonexistent elements before the first one and after the last, respectively.
* A LinkedList does this through the use of **nodes**. A node consists of two parts: the data that is being stored and pointers to the next/previous elements in the list.
* To find any element in the LinkedList, Java **must** start at either the beginning or the end of the list and traverse from element to element until it finds the element it's looking for. This can be bad for performance if you're searching the list often.
* Because LinkedLists aren't stored as one solitary unit, you can change (add, delete, or modify) any element you want with little overhead. Java does this by adding the new value and/or adapting **only** the nodes that point to the given element rather than having to copy the entire array as it does with ArrayLists. For example, if you insert something new at index three, only index two and index four will have to change their pointers–everything else will remain exactly the same.
* There are pros and cons to LinkedLists: Arrays are sometimes better, especially when we need to find an element at random. Because of this, it's a good idea to use ArrayList if you're creating a list that doesn't need to be changed frequently.
* However, if you need to manipulate data (adding, removing, or changing elements) frequently, a LinkedList will perform far better.
* As an implementation of the List interface, elements in a LinkedList **can be repeated**, and their **order** will always be maintained.
* Here's a handy diagram to help you visualize Java's LinkedList. Note that bottom portion of the image contains a LinkedList with three Strings, "How", "Lists", and "Work." You can trace their nodes below:



In the above, if we needed to add a new element in between "How" and "Lists," we'd change the "next" address pointer for "How" (the one that says 222) and the "previous" address pointer for "Lists" (the one that says 111), to match the new element. The "Work" node would remain completely untouched, since none of its neighbours has changed. This is how LinkedLists support quick and easy changes to their contained elements.

## Methods

LinkedList comes with many of the same methods as the ArrayList, but it also has a few of its own. Read through them below:

* int size()**:** returns the number of elements in the list.
* boolean isEmpty()**:** returns true if there are no elements in the list, or false if it contains one or more elements.
* boolean contains(Object o)**:** returns true if the specified element is found somewhere within the list.
* Iterator iterator()**:** returns an [iterator](https://www.w3schools.com/java/java_iterator.asp) that loops over the elements in this list in proper sequence starting from the first element.
* Object[] toArray()**:** returns an array containing all of the elements in this list in proper sequence.
* boolean add(E e)**:** appends the specified element to the **end** of the list.
* boolean remove(Object o)**:** removes the first occurrence of the specified element from this list.

InfoWarningTip

Keep in mind, LinkedLists can contain the same element multiple times, just like an ArrayList can. This is one of its primary identifying features.

* boolean retainAll(Collection c)**:** clears the list, but retains the elements in this list that are contained in the specified collection.
* void clear()**:** removes all elements from the list leaving it empty.
* E get(int index)**:** returns the specified element from the given position.
* E set(int index, E element)**:** replaces the element at the specified position in the list with the element you pass in.
* ListIterator listIterator()**:** returns a [ListIterator](https://www.digitalocean.com/community/tutorials/java-listiterator) over the elements in the list which can go backwards or forwards through the list and perform other functions as it goes.
* List subList(int fromIndex, int toIndex)**:** returns the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. Think of this as a way to get a set of values from the middle of the LinkedList.