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.

We already have a type of "collection" (kind of) in our understanding of Java: arrays. Have you noticed any restrictions/shortcomings with your arrays, though? What happens if you define an array to have only three indices, but you need four? Have you thought about what happens when you define an array with too many indices and don't use them? What happens then?
Arrays in Java have several limitations, including their fixed size, the need to know the size at declaration, potential memory wastage when unused slots are allocated, lack of built-in methods for common operations, manual management of resizing and

element copying, and limited support for generics. In contrast, ArrayLists provide dynamic sizing, automatic resizing, and a range of built-in methods for adding, removing, and manipulating elements, offering greater flexibility and convenience. However, ArrayLists may have slightly higher memory overhead and a small performance cost for certain operations compared to arrays. The choice between arrays and ArrayLists should consider the specific requirements of your program.

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** (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.

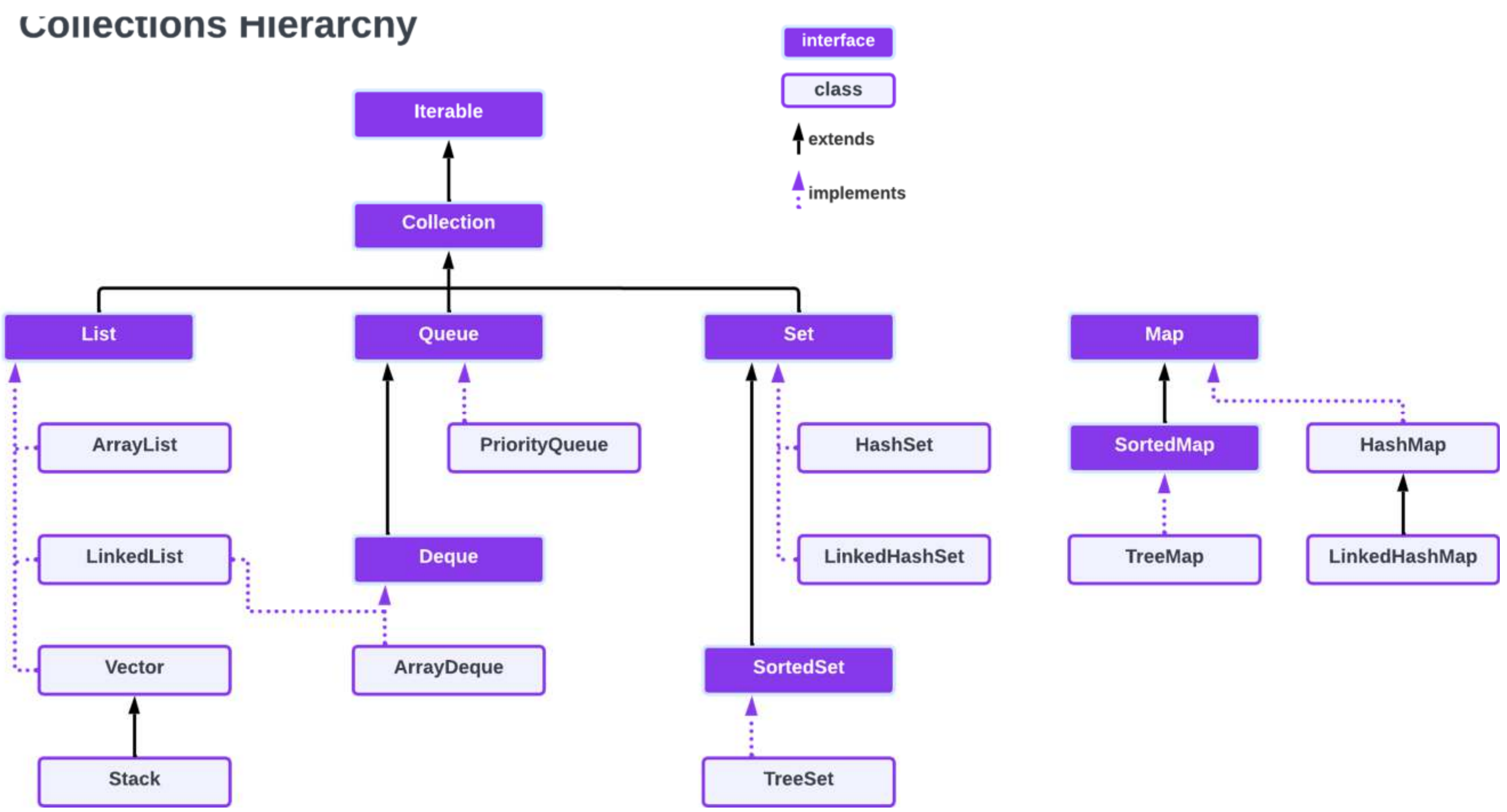
#The Java Collections Framework

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:



The hierarchy of interfaces and classes within the collections framework

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.

InfoWarningTip
As stated above, we'll be going in-depth on these interfaces and classes throughout the week. This is just an introduction, so keep on moving through your lessons, even if this doesn't feel like it has sunk in 100% of the way. They'll be easier to understand once you've played with them a little.

#Knowledge Check

Which of the following does not implement the collection interface, and it is used to store key/value pairs?

Like standing in a line, the _____ interface can work on a first-in, first-out methodology. As a result, we can add items from the start of the collection and remove items at the end of the collection.

Of the following, select all of the collections that are **unordered**:

What is the `java.util.Collections` **class**? Hint: Collections Java docs.