Storing Data Using the Collections Framework

Introducing the Collections Framework

The Collections Framework is the most widely used API of the JDK. Whatever the application you are working on is, odds are that you will need to store and process data in memory at some point.

The history of data structures goes back nearly as far back as computing itself. The Collections Framework is an implementation of the concepts on how to store, organize, and access data in memory that were developed long before the invention of Java. The Collections Framework does this in a very efficient way, as you are going to see.

The Collections Framework was first introduced in Java SE 2, in 1998 and was rewritten twice since then:

- in Java SE 5 when generics were added;
- in Java 8 when lambda expressions were introduced, along with default methods in interfaces.

These two are the most important updates of the Collections Framework that have been made so far. But in fact, almost every version of the JDK has its set of changes to the Collections Framework.

You will learn in this part are the most useful data structures the Collections Framework has to offer, along with the patterns you will be using to manipulate this data in your application.

The first thing you need to know is that, from a technical point of view, the Collections Framework is a set of interfaces that models different way of storing data in different types of containers. Then the Framework provides at least one implementation for each interface. Knowing these implementations is as important as the interfaces, and choosing the right one depends on what you need to do with it.

Finding Your Way in the Collections Framework

The amount of interfaces and classes in the Collections Framework may be overwhelming at first. Indeed, there are many structures available, both classes and interfaces. Some have self-explanatory names, like `LinkedList`, some are carrying behavior, like `ConcurrentHashMap`, some may sound weird, like `ConcurrentSkipListMap`.

You are going to use some of these elements much more often than others. If you are already familiar with the Java language, you probably came across `List`, `ArrayList` and `Map` already. This tutorial focuses on the most widely used structures of the Collections Framework, the ones you are going to use daily as a Java developer, and that you need to know and understand best.

That being said, you need to have the big picture of what is there for you in the Collections Framework.

First, the framework consists of interfaces and implementations. Choosing the right interface means you need to know what functions you want to bring to your application. Is what you need consists in:

- storing objects and iterating over them?

- pushing your object to a queue and popping them?
- retrieving them with the use of a key?
- accessing them by their index?
- sorting them?
- preventing the presence of duplicates, or null values?

Choosing the right implementation means you need to know how you are going to use these functionalities:

- Will accessing your objects be done by iterating, or random, indexed access?
- Will the objects be fixed at the launch of your application, and will not change much over its lifetime?
- Will the amount of objects be important with a lot of checking for the presence of certain objects?
- Is the structure you need to store your objects in will be accessed concurrently?

The Collections Framework can give you the right solution to all of these problems.

There are two main categories of interfaces in the Collections Framework: collections and maps.

Collections are about storing objects and iterating over them. The `Collection` interface is the root interface of this category. In fact, the `Collection` interface extends the `Iterable` interface, but this interface is not part of the Collections Framework.

A Map stores an object along with a key, which represents that object, just as a primary key represents an object in a database, if you are familiar with this concept. Sometimes you will hear that maps store *key/value* pairs, which exactly describes what a map does. The `Map` interface is the root interface of this category.

There is no direct relationship between the interfaces of the `Collection` hierarchy and the `Map` hierarchy.

Along with these collections and maps, you also need to know that you can find interfaces to model queues and stacks in the `Collection` hierarchy. Queues and stacks are not really about iterating over collections of objects, but since they have been added to the `Collection` hierarchy, it turns out you can do that with them.

There is one last hierarchy that you also need to know, which is the `Iterator` hierarchy. An iterator is an object that can iterate over a collection of objects, and it is part of the Collections Framework.

That makes two main categories, `Collection` and `Map`, a subcategory, `Queue`, and a side category, `Iterator`.

Avoiding Using Old Interfaces and Implementations

The Collections Framework was only introduced in Java 2, meaning that there was a life before it. This life consisted in several classes and interfaces that are still in the JDK, to preserve backward compatibility, but that you should not use anymore in your applications.

Those classes and interfaces are the following:

- `Vector` and `Stack`. The `Vector` class has been retrofitted to implement the `List` interface. If you are using a vector in a non-concurrent environment, then you can safely replace it with `ArrayList`. The `Stack` class extends `Vector` and should be replaced by `ArrayDeque` in non-concurrent environments.
- The `Vector` class uses the `Enumeration` interface to model its iterator. This interface should not be used anymore: the preferred interface is now the `Iterator` interface.

- `HashTable`: This class has been retrofitted to implement the `Map` interface. If you are using instances of this class in a non-concurrent environment, then you can safely replace its use with `HashMap`. In a concurrent environment, `ConcurrentHashMap` can be used as a replacement.

Why Choose a Collection Over an Array?

You may be wondering why you should bother learn the Collection Frameworks when you may have the feeling that putting your data in a good old array does the job.

The fact is, in any case, if you have a solution that is simple, that you master well, and that fits your needs, then you should definitely stick with it!

What can a collection do for you, that an array cannot?

- A collection tracks the number of elements it contains
- The capacity of a collection is not limited: you can add (almost) any amount of elements in a collection
- A collection can control what elements you may store in it. For instance, you can prevent null elements to be added
- A collection can be queried for the presence of a given element
- A collection provides operations like intersecting or merging with another collection.

This is just a small sample of what a collection can do for you. In fact, since a collection is an object and given that an object is extensible, you can add any operation you need on most of the collections provided by the JDK. It is not possible to do this with an array, because an array is not an object in Java.

Last update: *September 14, 2021*

Storing Data Using the Collections Framework

Getting to Know the Collection Hierarchy