GE Research Report: The Java Collections Framework

By Joel Pillar-Rogers (word count: 1832)

Introduction

The Java Collections Framework is a fundamental part of the Java programming language. It is largely concerned with collections, which are objects that group other objects, and algorithms that operate on the collections. The Framework was released in 1998 with the intention of unifying the data structures in Java, making them easy to use, light-weight and efficient, with a generic interface. The C++ Standard Template Library was a similar attempt to unify the data structures in that language. The two collections frameworks have many things in common, which is demonstrated by an implementation of a program in both languages.

History

The Java Collections Framework was released by Sun Microsystems (later Oracle) as part of Java 2, Standard Edition 1.2 in 1998 (Blosch 2016; Zukowski 2008). It replaced or extended several uncollected data structures in core Java, such as Arrays, Vectors, Stacks, Enumeration and HashTables (Blosch 2005, 2016; Singh 2016; Zukowski 2008). These structures had no common interface and had different methods for manipulating the data, making it difficult to write algorithms to operate across the different structures (Singh 2016). The Collection Framework also replaced some commonly used data structure packages from external customised libraries, such as JGL and the *collections* package by Doug Lea, which were attempts to provide Java with some of the functionality of the C++ Standard Template Library (Blosch 2016; Lea 1999; Vanhelsuwé 1999). The latter would form the basis of Java's Collection Framework. The Framework was largely written by Joshua Blosch, who at the time was Senior Staff Engineer in the Java Software division (Blosch 2016; Sun Microsystems 2005).

Basic Principles

The Java Collections tutorial describes a Collection Framework as:

> *A unified architecture for representing and manipulating collections (Oracle 2017).*

A collection is an object that groups other objects or primitives. It stores and retrieves these data, and allows modifying, sorting and searching of them with operations that are independent of the underlying data structure (Lewis 2019; Oracle 2017).

The Java Collections Framework was designed to provide a lightweight yet powerful set of data structures with unified and familiar-feeling interfaces (Blosch 2016; Oracle 2019b). Collections were designed to have as few methods as possible and only where there were “truly basic” operations. They were also designed to integrate with existing data structures (Blosch 2016; Oracle 2019b).

The Framework contains three key elements: **Interfaces**, **Implementations** and **Algorithms** (Oracle 2017). There are four base interfaces: *Collection*, *List*, *Set* and *Map*. These cannot be instantiated and must be extended by other interfaces or implemented by classes. They designate methods that all implementing classes must define, so programmers know they can be called on any subclass (Oracle 2019c).

Some of the common classes to implement Collection are ArrayList, LinkedList, HashSet and TreeSet. HashMap and TreeMap implement the Map interface, and are not true descendants of Collection, but share many of the same methods.

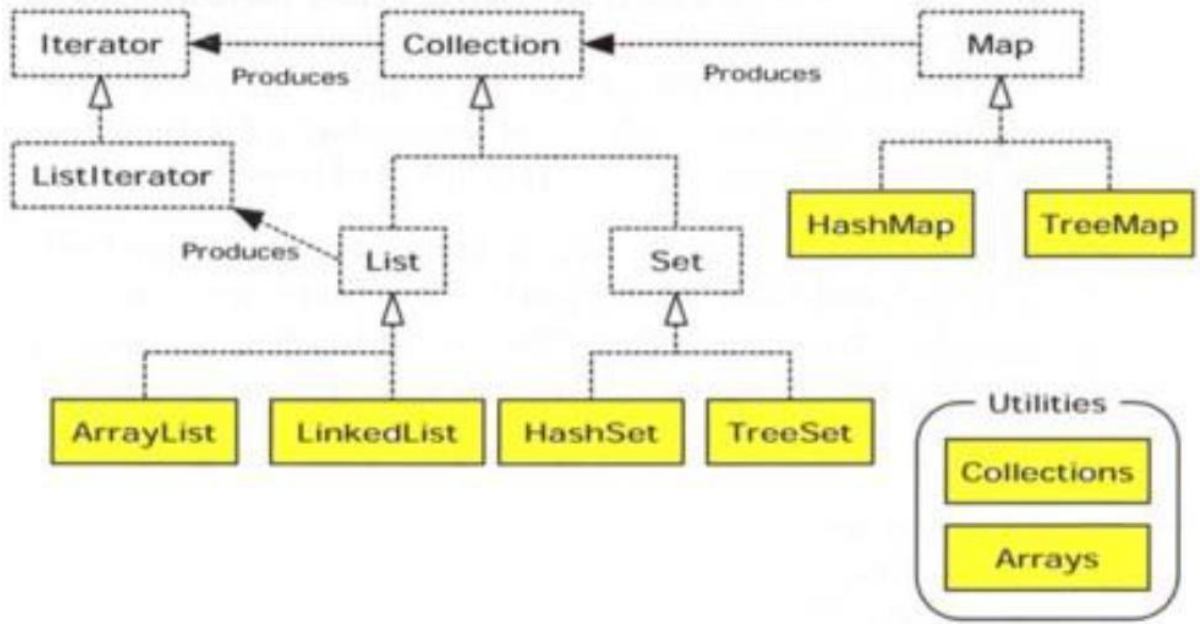

Figure 1. Collections Map (Lewis 2019)

There are two utility classes which provide algorithms to manipulate collections, called Collections and Arrays. Java Collection classes are Generic, which means they are able to hold all kinds of objects, but you must specify which type on instantiation (Oracle 2019c).

Benefits

There are many benefits to designing a unified collections framework. The Framework provides useful data structures with well documented methods that reduce the need for programmers to rewrite basic structures. The collections are high-quality and high-speed implementations of the data structures, so programmers know they are getting efficient code. APIs that implement collections are "interoperable", so can interact despite being written by different coders. The methods to use collections are standardised so there is no need to learn lots of different APIs and it is easier to design new APIs that conform to the Collections Framework. Finally, new data structures and algorithms that conform to the Collections Framework can be easily reused by other programmers (Becker 1999; Blosch 2005, 2016; Lewis 2019; Singh 2016).

Comparison with C++ Standard Template Library (STL)

The C++ Standard Template Library has containers which are quite similar to collections in the Java Collections Framework. The Standard Template Library was largely devised by Alex Stepanov and released in 1994, four years before Java's Collections Framework (Stevens 1995). Stepanov recognised that algorithms could be abstracted from a particular data structure, to be made generic across multiple data structures, yielding a large efficiency gain.

The STL has a similar structure to the Collections Framework. It has **containers**, **adaptors**, **algorithms** and **iterators** (Stepanov & Lee 1995). The *containers* are similar to the *interfaces* of Java although they can be directly implemented. However, they also serve as the basis for the *adaptor* classes, which are similar to Java *implementations*. C++ *algorithms* are similar to Java *Collections*. C++ further divides containers into sequence containers, like *lists* and *vectors*, and associative containers, like *sets* and *maps* (Stepanov & Lee 1995). Iterators are frequently used in STL containers. Each container has built-in iterators for traversing its members, and which are also called by algorithms. Much of this iterator behaviour is hidden behind the methods of collections and the Collections class in Java.

Similar to Java collections, the C++ containers are generic (called templates) in that they can hold all kinds of objects.

How does C++ represent Lists, Sets, and Maps?

C++ had several representations of these three Java interfaces. For Java's List, C++ also has *list*, as well as the other sequence containers including vector and array. C++ has a set and map, to match those collections in Java (cplusplus 2019b).

category	Java collection	C++ container
sequence	List	*(no direct comparison)*
	Array	array
	ArrayList	vector
	Linked List	list
	Queue	queue
	Deque	deque
associative	Set	*(no direct comparison)*
	HashSet	unordered_set
	TreeSet	set
	Map	*(no direct comparison)*
	HashMap	unordered_map
	TreeMap	map

Table 1. Comparison of some Java collections and C++ containers (cplusplus 2019b)

How does C++ deal with sorting?

In Java, sorting is handled by the Collections class. Sort can work either on the natural ordering of objects in a collection or can accept a Comparator object as a parameter (Oracle 2019a). Behind the scenes, sort is usually a mergesort or a modified quicksort.

In C++, sorting is handler by the <*algorithm*> collection of functions. There are several versions of sort(), depending on whether you want the order of equal objects maintained or only want to partially sort objects. C++ sort() accepts a beginning and ending iterator as parameters and sorts the range between them. Similar to Java, C++ sort() can accept a compare function or a class that provides a Boolean value to indicate the order of two objects (cplusplus 2019a). Implementations of C++ sort() can differ, but roughly conform to $n\log_2(n)$, indicating the use of quicksort, mergesort or some variation.

Benefits or drawbacks between the different collections?

The Java collections are easier to understand and use, the structure of the interfaces and implementation is clear, and the collections often seem to have ready methods for all the basic programming tasks. The Java API is very helpful and has detailed descriptions of the collections. By comparison, C++ STL containers are clunky and require more in-depth knowledge of the language and programming to use safely. On the other hand, C++ containers and algorithms seem to give more control over your program.

The frequent use of iterators in C++ is a clear difference on the surface. This is evident in the sorting methods, which Java handles in a cleaner way, although this results in fewer options. Iterators can get confusing for novices in C++ while iterators in Java are generally handled behind the scenes.

Implementation of Dog and Dog Register in C++

I used the set, vector and algorithm classes from C++ to mimic the HashSet, ArrayList and Collections classes used by my Java (DogBreeder) application. My submitted C++ files are all executable.

Dog Class

The Dog class from Practical 2 stores the details of individual dogs. Most of the members are simple strings and integers, however, one member is a Set, implemented as a HashSet, which stores the details of the dog's owners as strings. Owners are added to the HashSet using the method *add()*, which is inherited from the Collection interface.

The basic strings and ints are easily reproduced in C++. I #included <set> to implement the owner collection in C++ and I used the built-in *insert()* function of sets to mimic the *add()* method of HashSets.

Java syntax:

```
import java.util.HashSet;
private Set<String> owners = new HashSet<>();
public void addOwner(String newOwner)
{
   owners.add(newOwner);
}
```

C++ syntax:

```
#include <set>
std::set<std::string> owners_;
void Dog::addOwner(string newOwner)
{
   owners_.insert(newOwner);
}
```

DogRegister class

The DogRegister class stores dog objects in an ArrayList, which is an implementation of the List interface. It uses the *add()* method inherited from the Collection interface, and the *get()* method inherited from the List interface. DogRegister uses algorithms from the Collections class to sort the dogs by breed, with an implementation of the Comparator interface. Finally, DogRegister uses polymorphism to implement selecting dogs by condition.

I #included <vector> to mimic the ArrayList and I used the built-in *push_back()* method to add dogs to the end of the list. I used the built-in *erase()* function to delete dogs, although this required the use of iterators as there were no delete methods that took index numbers.

Java syntax:

```
import java.util.ArrayList;

ArrayList<Dog> DogRegister = new ArrayList<>();

public void addDog(Dog dog)
{
   DogRegister.add(dog);
}

public void deleteDog(int regNum)
{
   for (int i = 0; i < DogRegister.size(); i++)
   {
      if (DogRegister.get(i).getRegNum() ==
regNum)
      {
         DogRegister.remove(i);
      }
   }
}
```

C++ syntax:

```
#include <vector>

std::vector<Dog> register_;

void DogRegister::addDog(Dog dog)
{
   register_.push_back(dog);
}

void DogRegister::deleteDog(int regNum)
{
   for (int i = 0; i < register_.size(); i++)
   {
      if (register_.at(i).getRegNum() == regNum)
      {
         register_.erase(register_.begin() + i);
      }
   }
}
```

The DogRegister class also contained a sort method, based on Dog breeds, called groupByBreed(). In Java, this used a Collections method to operate on the ArrayList. It also used a separate class, DogSorter, which implemented the Comparator class to compare dogs.

In C++, I used the <algorithm> *sort()* on the vector of Dogs. To mimic the Java Comparator implementation, I made a new method that returned a boolean for which dog's breed was lexicographically first. The C++ *sort()* method requires iterators to determine the range of the sort, for which I used vector's built-in *begin()* and *end()* methods.

Java syntax:

```
import java.util.Collections;

public void GroupByBreed()
{
   Collections.sort(DogRegister, new DogSorter());
}

public class DogSorter implements
Comparator<Dog>
{
   public int compare(Dog d1, Dog d2)
{
   return
d1.getDogBreed().compareTo(d2.getDogBreed());
   }
}
```

C++ syntax:

```
#include <algorithm>

void DogRegister::groupByBreed()
{
   sort(register_.begin(), register_.end(),
compareDogs);
}



bool DogRegister::compareDogs(Dog d1, Dog d2)
{
   return
d1.getDogBreed() < d2.getDogBreed();
}
```

Finally, the DogRegister class had a couple of methods to return a list of Dogs who fulfilled certain criteria. The first was dogs whose names contained a character sequence, using the *contains()* method of the String class. The second was dogs who fulfilled a condition set in a separate class. The classes that could fulfil these conditions all had to implement the DogCondition interface, and a polymorphic reference was used for DogCondition in the method parameters.

In C++, for the character sequence method I used the *find()* function, which is a built-in function of C++ strings, but which returns an index number to the start of the character sequence instead of a boolean value.

For the condition method, I created a new abstract class (.h file) called DogCondition that contained a single virtual method, *satisfies()*. I then added a class to DogRegister.cpp that implemented this class. Finally, I used a polymorphic reference to DogCondition in the method parameter. This turned out very similar to the Java implementation, but it took a lot longer to work out the correct syntax of C++.

For both methods, in C++ I chose to return a DogRegister instead of a vector (in Java I returned an ArrayList). This is simply because ArrayLists allow printing of their values and vectors don't, so I used the built-in print method of DogRegister.

Java syntax:

```
public ArrayList<Dog>
getDogsWhoseNameContains
(String charSequence) {
   ArrayList<Dog> dogNames = new ArrayList<>();
   for (int i = 0; i < DogRegister.size(); i++)
{
      if (DogRegister.get(i).getName()
          .contains(charSequence))
{
         dogNames.add(DogRegister.get(i));
      }
   }
   return dogNames;
}

public ArrayList<Dog>
getByCondition(DogCondition c)
{
   ArrayList<Dog> dogList = new ArrayList<>();
   for (int i = 0; i < DogRegister.size(); i++)
{
      if (c.satisfies(DogRegister.get(i)))
{
         dogList.add(DogRegister.get(i));
      }
   }
   return dogList;
}
```

C++ syntax:

```
DogRegisterDogRegister::
getDogsWhoseNameContains
(string charSequence)
   DogRegister dogNames;
   for (int i = 0; i < register_.size(); i++)
   {
      if (register_.at(i).getName()
          .find(charSequence) != string::npos)
      {
         dogNames.addDog(register_.at(i));
      }
   }
   return dogNames;
}

DogRegister DogRegister::
getByCondition(const DogCondition &c)
{
   DogRegister dogList;
   for (int i = 0; i < register_.size(); i++)
   {
      if (c.satisfies(register_.at(i)))
      {
         dogList.addDog(register_.at(i));
      }
   }
   return dogList;
}
```

```java
public interface DogCondition
{
   public boolean satisfies (Dog d);
}
```

```cpp
class DogCondition
{
virtual bool satisfies(Dog d) const = 0;
};
```

```java
public class DogsNamedAtti
implements DogCondition
{
public boolean satisfies(Dog d)
   {
   boolean satisfied = d.getName() == "Atti";
   return satisfied;
   }
}
```

```cpp
class DogsNamedAtti
: public DogCondition
{
bool DogsNamedAtti::satisfies(Dog d) const
  {
  bool satisfied = d.getName() == "Atti";
  return satisfied;
  }
}
```

I included a *main* method to test the implementation of my classes in C++. I kept the setup very similar to the Java *main* method. This is included in the files provided.

Conclusion

The Java Collections Framework and the C++ Standard Template Library present a unified architecture of data structures for their individual languages. The generic interfaces of the collections allow algorithm to operate on them without needing to know the details of their implementation. This makes programming more efficient and encourages reuse of software. The basic design of the Framework and Library is the same: core collections, collections which extend these core collections and algorithms to operate on them. The STL makes frequent use of iterators, whereas the Framework mostly uses iterators behind the scenes. In general, the two architectures are quite similar, as shown through an implementation of the DogBreeder program.

References

Becker, D 1999, *Get started with the Java Collections Framework*, JavaWorld, viewed 3 June 2019, <https://www.javaworld.com/article/2076800/get-started-with-the-java-collections-framework.html>.

Blosch, J 2005, *Trail: Collections*, Sun Microsystems, viewed 3 June 2019, <http://www.iitk.ac.in/esc101/05Aug/tutorial/collections/index.html>.

—— 2016, *The Design of the Collections API – Parts 1 & 2*, lectures slides, delivered October 2016, 15-214 Principles of Software Construction, Carnegie Mellon University.

cplusplus 2019a, *Standard Template Library: Algorithms*, viewed 4 June 2019, <http://www.cplusplus.com/reference/algorithm/>.

—— 2019b, *STL Containers*, viewed 3 June 2019, <http://www.cplusplus.com/reference/stl/>.

Lea, D 1999, *Overview of the collections Package*, State University of New York at Oswego, viewed 3 June 2019, <http://gee.cs.oswego.edu/dl/classes/collections/index.html>.

Lewis, T 2019, *The Collections Framework*, lecture slides, delivered 22 March 2019, COMP8741 Application Development, Flinders University.

Oracle 2017, *The Java Tutorials: Collections*, viewed 3 June 2019, <https://docs.oracle.com/javase/tutorial/collections/index.html>.

—— 2019a, *Class Collections*, Oracle, viewed 4 June 2019, <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>.

—— 2019b, *Collections Framework Overview*, viewed 3 June 2019, <https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>.

—— 2019c, *Inteface Collection<E>*, Oracle, viewed 3 June 2019, <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>.

Singh, D 2016, *Collections in Java*, GeeksforGeeks, viewed 3 June 2019, <https://www.geeksforgeeks.org/collections-in-java-2/>.

Stepanov, A & Lee, M 1995, *The Standard Template Library*, HP Laboratories

Stevens, A 1995, *Interview with Alex Stepanov*, Dr. Dobb's Journal, viewed 4 June 2019, <http://stepanovpapers.com/drdobbs-interview.pdf>.

Sun Microsystems 2005, *Bios for Contributing Authors*, viewed 3 June 2019, <http://www.iitk.ac.in/esc101/05Aug/tutorial/information/bios.html#jbloch>.

Vanhelsuwé, L 1999, *The battle of the container frameworks: which should you use?*, JavaWorld, viewed 3 June 2019, <https://www.javaworld.com/article/2076326/the-battle-of-the-container-frameworks--which-should-you-use-.html>.

Zukowski, J 2008, *Java Collections*, Apress L. P., Berkeley.