Companion Chapter 5

Java Collections Framework

What You Will Learn in This Chapter

The Java Collections framework contains class implementations of data structures that developers can find incredibly useful in many information systems and application contexts. Industry-standard data collection approaches such as resizable arrays, stacks, queues, sets, and maps (with hashing) are all implemented through the classes of the framework. Developers who want to take advantage of these data-handling approaches in memory can use the Java implementations, which have been highly optimized for speed and efficiency, without having to implement their own versions. Much of the functionality of these classes will also perform other complex data-organizing tasks for the developer, saving them even more time to focus on the major business requirements of an information systems implementation instead. These collection classes offer a great deal of flexibility and power in easy-to-use implementations.

Specifically, this chapter will help you do the following:

1. Learn about the nature of the Java Collections framework and its use of interface classes and abstract methods to implement interoperability across the classes of the framework
2. Learn how to fully use the ArrayList<E> class and leverage features of the Collections framework and its methods
3. Learn about the nature of stacks, queues, and sets; their specific implementations in the Java Collections framework; and how to create and use them effectively
4. Learn about the nature of the Iterable<T> interface, Iterator<E> class, and how these provide a common way for Java code to interact with many of the Collections classes
5. Learn about the nature of a map data structure and its specific implementations in Java and how to create and use them effectively
6. Understand how to apply many of these data structures to solve real-world, business-related problem scenarios

Opening Scenario

"Speed and security. We must be able to demonstrate to the client that the system is capable of efficient data handling and processing"—your project lead for the mom-and-pop grocery store IS project started off the most recent team meeting with this proclamation. A recent meeting with the client yielded stories of how they observed "slow and buggy" software packages from other vendors, vendors who are currently trying to lure them away from your team's efforts.

Taking these fears to heart, you begin discussions with your colleagues on the implementation team. "We should take advantage of the data structures built into the language for handling in-memory data like inventory and so on," one team member commented. In your research on creating efficient searching of inventory items, you skimmed the documentation for the ArrayList<E> class in Java but noticed discussion of a "Collections framework" that offers much more than just ArrayList<E>. Backing this up is the research you have done showing that some Java classes have been implemented that are "highly efficient" at arranging and retrieving data from array-like objects. This certainly sounds like the right direction to go in giving the system "speed."

You feel confident that a full understanding of what each major class in the framework does and its

Copyright © 2024 Prospect Press. All rights reserved. For use with *Fundamentals of Java Programming for Information Systems* by Jeremy D. Ezell

advantages over simple arrays will help a great deal in keeping in-memory data processing speedy and efficient. The team feels that demonstrating these data structures to the client would be a big step toward alleviating their fears. After wrapping up some other areas of research and functionality, you once again dive into Java documentation, hoping to fully understand the classes of the Java Collections framework . . .

CC5.1 Classes of the Java Collections Framework

A collection is simply a grouping of data that shares some common nature and is organized in a purposeful and useful manner. You have worked with a collection in a basic way through the use of arrays (see chapter 7). For example, a list of the names of cities could be considered a collection. In most programming languages, the ability to create and work with a collection has been accomplished through the use of what are called data structures. An array is a data structure, for example. These are usually objects (i.e., "complex data type") with an internal structure that is designed for the storage and processing of a collection (i.e., sequence) of data.

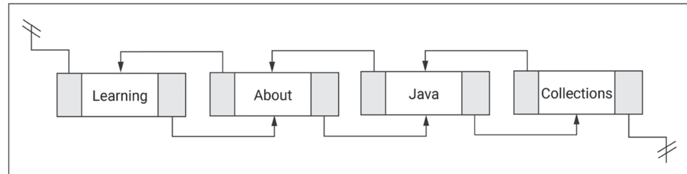

Figure CC5.1. Visual of a Java LinkedList<String> Data Structure

The Java Collections framework is a family tree of interfaces, abstract classes, and their concrete implementations (see companion chapter 4 for the basics on these class types) that enables developers to create and use sophisticated data structures to store and process data. The Collections framework emerged from what was previously just a grouping of classes that, while officially unrelated to one another, shared a common data-handling intent. By redesigning older classes like Vector, Stack, Properties, and Dictionary; creating new classes; and connecting various "family trees" of these old and new classes through the use of interfaces, the Collections framework emerged. Oracle's technical note documentation regarding the Collections framework mentions its many benefits: unifying classes that are intended to work with collections of data under a common API through interfaces, providing computationally efficient implementations of these data structures in Java preventing developers from having to "roll their own" versions, using object-oriented principles in the design of the Collections classes to ease their use for new developers, and increasing the interoperability and extensibility of these classes, among other benefits.[*]

Within the field of information systems, collections of data are very common. Typically these are stored in a database table, but there are often occasions when data needs to be manipulated in memory prior to a database write operation. The reverse is also true: often, data is read in from a database and manipulated in memory during system usage. Lists of usernames and user accounts, lists of project tasks, and collections of inventory items are common collections dealt with among many others in business contexts. So it is beneficial to explore the Java Collections framework and see how the data structures implemented within it work and how they can be useful to us. Figure CC5.2 shows a partial map of the relationships between the classes in the Java Collections framework.

ArrayList<E> and methods of the Collection<E> interface: You have already been introduced to and worked with the ArrayList<E> class several times in this textbook, so it is a great starting place for our exploration of the Collections framework. Notice that in figure CC5.2, the interface class Collection<E> is the top-level class of most of the Collections framework.[†] Be careful not to confuse the Collection<E> interface with the utility class Collections, which is also a member of the Java Collections framework.[‡] The latter contains mostly static methods that provide convenient operations on instance objects of classes in the Collections framework (like ArrayList<E>, which we worked with in previous chapters). The Map<K,V>

[*] https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/doc-files/coll-overview.html.
[†] https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Collection.html.
[‡] https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Collections.html.

Figure CC5.2. Partial, Simplified Map of the Java `Collections` Framework

interface class serves as the top-level class for the "second half" of the `Collections` framework, providing a common structure and behavior for a more complex data structure that we will discuss later in the chapter.

As you can see in figure CC5.2, there are several subinterfaces to `Collection<E>` and `Map<K,V>`. Implementing these are numerous abstract classes, and inheriting from these with concrete implementations are numerous concrete classes. For many examples that you will explore in this chapter, the interface will occasionally be used as the reference type and the concrete class as the instantiated type (see companion chapter 4 for details on how interfaces can be used in this manner). For example, consider the two following instantiations of an `ArrayList<E>` object:

Code Snippet CC5.1

```
List<Integer> myArrayList = new ArrayList<>();

ArrayList<Integer> myArrayList2 = new ArrayList<>();
```

Both are valid. Since `ArrayList<E>` implements the `List<E>` interface, the interface can be used as the reference type for the `ArrayList<E>` instance object. But the second line of code is more common for `ArrayList<E>`.

You might notice a type parameter being used, as these interfaces and classes are generic type (see companion chapter 4 for details on generic classes). This gives developers flexibility to use any type of values or objects in the data structures created by instantiating these classes. Remember that only a complex data type can be used for a type parameter, whereas primitives cannot. Java conveniently provides wrapper classes (all of them subclasses of class `Number`) and provides the autoboxing/autounboxing functionality to convert between primitives and their wrapper class versions for you.[*] So if you want an `ArrayList<E>` of `int` values, you will need to use the wrapper class `Integer`. Java will perform the conversion for us.

[*] https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Number.html.

The following are some overall features of the Collections framework to keep in mind:

- The data structures created by the classes of the Collections framework are dynamically resizable. They will grow and shrink as you add to and remove elements from them.
- Since interfaces provide a commonality across these classes but do not dictate functionality, all subinterfaces and classes of the Collections framework are able to implement various tasks like adding, removing, indexing (or not), and so on for their data structures in ways that are unique to their purpose. They may share functionality with other Collections classes but implement it in different and unique ways.
- As you have learned, since interfaces unify the Collections classes in the framework, any Java code that expects a parameter type of one of the interfaces can use objects from across the Collections framework classes.
- The classes in the framework implement industry-standard and algorithmically efficient behaviors for their data structures, such as adding, removing, searching, and sorting. You do not have to code these on your own or worry about low-level details for handling the data in the structures, freeing you to focus on the larger problem of the system being implemented.

For the classes that implement the Collection<E> interface, there are several methods commonly and concretely implemented across them. Keep in mind that these are uniquely implemented in each class and may behave differently in each. Table CC5.1 briefly describes some of these common methods shared across the Collections framework.

Remember that these methods are defined as abstract methods in the Collection<E> interface. It is up to abstract and concrete classes lower on the "family tree" of the Collections framework to implement these concretely, and they all do. Some of the subinterfaces that implement Collection<E> add to these methods

Table CC5.1. Partial Listing of Instance Methods of the Collection<E> Interface

Instance method signature	Description
boolean .add(E element)	Adds the element of type E to the data structure. Some classes return a boolean value indicating successful addition (true) or not (false).
void .clear()	Completely empties the data structure.
boolean .contains(Object obj)	Tests whether the data structure contains the object referenced by the method parameter or not. Returns true or false accordingly.
boolean .containsAll(Collection<?> coll)	Tests whether or not the data structure the method is invoked upon contains all of the elements of the data structure passed in via reference parameter. Returns true or false accordingly.
boolean .isEmpty()	Returns true if the data structure contains no elements. Returns false otherwise.
Iterator<E> .iterator()	Returns a reference to an iterator instance object that can traverse (i.e., "walk") through the elements of the data structure, providing a convenience to developers.
boolean .remove(Object obj)	If present, the object that matches the one whose reference is passed in as a parameter will be removed from the data structure. A boolean value is returned indicating the success/failure of this operation.
boolean .removeAll(Collection<?> coll)	Removes from the data structure all the objects that match the ones in the parameter data structure.
boolean .retainAll(Collection<?> coll)	Keeps all the objects in the data structure that match the objects in the parameter data structure and removes all the rest that **do not** match.
int .size()	Returns the size of the data structure.
Object[] .toArray()	Creates a "standard" array that contains the elements of the data structure and returns a reference to the new array.

based on their specific nature and functionality. Since `ArrayList<E>` implements the `List<E>` interface and inherits from several abstract classes (not listed in figure CC5.2), it has additional methods beyond those listed in `Collection<E>`.*

The usage of some of the methods in table CC5.1 can be explored as follows by testing some code in a `main()` method Java application (**make sure** to import `java.util.*` first at the top of your code listing!):

Code Snippet CC5.2

```
ArrayList<Integer> myArrayList = new ArrayList<>();

myArrayList.add(1);
myArrayList.add(2);
myArrayList.add(3);

for (int num: myArrayList)
{
    System.out.print(num + " ");
}
System.out.println();

// Does the ArrayList contain the number 2?
System.out.println("Contains 2? "
        + myArrayList.contains(2));

// Is the ArrayList empty?
System.out.println("Is myArrayList Empty? "
        + myArrayList.isEmpty());

ArrayList<Integer> secondList = new ArrayList<>();
secondList.add(2);
secondList.add(3);

// Does the ArrayList contain all the elements found
//  within secondList?
System.out.println("Is secondList contained within myArrayList? "
        + myArrayList.containsAll(secondList));

// Add two numbers at index position 1 in the ArrayList
myArrayList.add(1, 42);
myArrayList.add(1, 56);

// Remove all elements from myArrayList that match the
//  ones found in secondList
System.out.println("Remove secondList from myArrayList success?: "
        + myArrayList.removeAll(secondList));

// Print the resulting contents of myArrayList
for (int num: myArrayList)
{
    System.out.print(num + " ");
}
System.out.println();
```

Running this code will yield the following output:

```
1 2 3
Contains 2? true
Is myArrayList Empty? false
```

* https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/ArrayList.html#method-summary.

```
Is secondList contained within myArrayList? true
Remove secondList from myArrayList success?: true
1 56 42
```

Notice that for many of the methods like `.contains()` and `.removeAll()`, you could probably implement the code for yourself. But luckily the developers of the Java language have done this for you, freeing you up to work on larger issues. Methods like `.removeAll()` or `.retainAll()` can be especially useful in filtering lists of data like usernames, products, locations, and so on.

The default behavior for an `ArrayList<E>` is to always add to the end of the list. Objects of the class are also capable of indexing, meaning you can insert and retrieve elements in the data structure by using index notation to identify a position in the list, same as with arrays:

Code Snippet CC5.3

```
myArrayList.add(1, 100);
int value = myArrayList.get(2);
```

Not all the classes in the `Collections` framework are capable of indexing. Keep in mind that `ArrayList<E>` does not use index notation as its primary element positioning behavior; it only provides it as a convenience to the developer. Figure CC5.3 shows visually what is happening in `myArrayList` when we add the number `100` at index position `1`.

You can use the `ArrayList<E>` method `.get()` with an index `int` as a parameter to retrieve an element from a particular index position in the data structure. Since all `Collections` framework classes store complex data-type objects, `.get()` returns back a reference to the object at that index position.

In addition, you can add duplicate values to an `ArrayList<E>` without any issue:

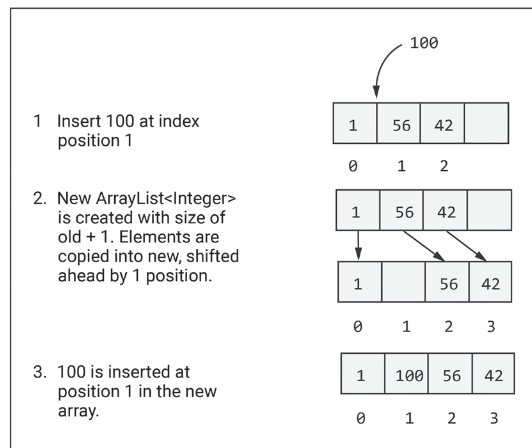

Figure CC5.3. Visual Behavior of Using `.add()` with an Index Position

Code Snippet CC5.4

```
myArrayList.add(316);
myArrayList.add(316);
myArrayList.add(316);

// Print the resulting contents of myArrayList
for (int num: myArrayList)
{
    System.out.print(num + " ");
}
System.out.println();
```

Running this additional code in our application will print the following:

```
. . . // Prior output from earlier.
1 100 56 42 316 316 316
```

Other classes in the `Collections` framework do not allow duplicates. To help with your understanding of the nature of some of these classes and to help with our discussion in the rest of this chapter, table CC5.2 summarizes some of the major characteristics of the concrete `Collections` framework classes we will explore:

Using the static methods of the `Collections` class: Recall earlier that a difference was pointed out between the `Collection<E>` interface that serves as the root-level interface for most of the Java `Collections` framework and the `Collections` class that contains only static methods. The methods of the `Collections` class provide a further convenience to developers who need to perform additional data maintenance actions on

Table CC5.2. Usage Characteristics of Some Collections Framework Classes

Collections class	Allows duplicates?	Uses index notation?	Insertion/removal action
ArrayList<E>	Yes	Provides but not core	Ordered as inserted
LinkedList<E>	Yes	Provides but not core	Ordered as inserted
PriorityQueue<E>	Yes	No	First in, first out (FIFO)
Stack<E>	Yes	No	Last in, first out (LIFO)
HashSet<E>	**No**	Provides but not core	Ordered by hash code of element
HashMap<K,V>	**No**	No	Ordered by hash code of key K
TreeSet<E>	**No**	No	Natural ordering based on value

collections without having to implement the often complicated algorithms involved themselves. Table CC5.3 describes some of the more commonly used static methods from the Collections class.[*] (*Note:* Some method signatures have been simplified using "<…>" for the sake of presentation in the table. Please see the official Oracle documentation for the full signature with complete type parameter.)

Note that some of the methods in table CC5.3 will only work with Collections objects whose classes have implemented List<E>, limiting your usage of these methods to that "branch" of the Collections family tree (see figure CC5.2).

Using a Java main() class application, you can explore the usage of these as follows. First, add a static method (see chapter 6) to the main() class application that looks like the following:

[*] https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Collections.html.

Table CC5.3. Selected Static Methods of the Collections "Utility" Class

Static method of Collections	Works with classes that implement _____ and those below it	Description
static <…> void **sort**(List<T> list)	List<E>	Changes the data structure referenced by parameter by sorting its elements by natural order.
static <…> int **binarySearch**(List<…> list, T key)	List<E>	Searches the data structure to find a match for the key, returning the index position if found. Elements must be sorted first for method to work effectively.
static void **reverse**(List<?> list)	List<E>	Changes the data structure by reversing the ordering of the elements contained within.
static void **shuffle**(List<?> list)	List<E>	Changes the data structure by shuffling its elements into a random ordering.
static <T> void **copy**(List<…> destination, List<…> source)	List<E>	Copies the elements found in the source data structure parameter into the destination data structure parameter, making changes to the destination collection.
static <T> void **fill**(List<…> list, T obj)	List<E>	Replaces all existing elements of the parameter data structure with copies of the parameter obj.
Static <…> T **max**(Collection<…> coll)	Collection<E>	Returns a copy of an element of the data structure with the highest "value" according to the natural ordering of those elements.
Static <…> T **min**(Collection<…> coll)	Collection<E>	Returns a copy of an element of the data structure with the lowest "value" according to the natural ordering of those elements.
static int **frequency**(Collection<?> coll, Object obj)	Collection<E>	Returns the count of the elements in the parameter data structure that match the parameter obj.

Code Snippet CC5.5

```
// Author's Print Method - You are free to use it!
public static void printCollection(Collection<?> dataStructure)
{
    for (Object obj: dataStructure)
    {
        System.out.print(obj + " ");
    }
    System.out.println();
}
```

This `printCollection()` method will make life easier as you print the results of invocations to the static methods of the `Collections` class. Notice that the methods accept a reference to any object whose class has implemented the Collection interface and is given the wildcard type parameter of "?" to allow any type (see companion chapter 4). Then a `for-each` loop is used to traverse (or "walk") the data structure and print its contents to the console.

***Note:* `printCollection()` will be used several times throughout the rest of the chapter.**

Next, we can invoke the methods of the `Collection` class in the demonstration:

Code Snippet CC5.6

```
ArrayList<String> nameList = new ArrayList<>();

nameList.add("Billi");
nameList.add("Jorge");
nameList.add("Sarah");
nameList.add("Zari");
nameList.add("Andi");
nameList.add("Yemik");
nameList.add("Andi");

// Print the list first
printCollection(nameList); // Author's print method

// Shuffle the elements
Collections.shuffle(nameList);
printCollection(nameList);

// Reverse this order
Collections.reverse(nameList);
printCollection(nameList);

// Find the "max" word
// (Lexicographically)
System.out.println(Collections.max(nameList));

// Find the "min" word
// (Lexicographically)
System.out.println(Collections.min(nameList));

// Sort the ArrayList
Collections.sort(nameList);
printCollection(nameList);

// Search the ArrayList now that it is sorted
System.out.println(Collections.binarySearch(nameList, "Sarah"));

// Determine the frequency of the appearance of a word
System.out.println(Collections.frequency(nameList, "Andi"));
```

This will produce the following output when run (with notes added by the author):

```
Billi Jorge Sarah Zari Andi Yemik Andi      (Initial List)
Billi Andi Andi Jorge Yemik Zari Sarah      (Shuffled)
Sarah Zari Yemik Jorge Andi Andi Billi      (Reversed)
Zari                                        (Max word - Z)
Andi                                        (Min word - A)
Andi Andi Billi Jorge Sarah Yemik Zari      (Sorted)
4                                           (Index Position of "Sarah")
2                                           (Count of "Andi")
```

Using the `LinkedList<E>` class: Closely related to the `ArrayList<E>` class is `LinkedList<E>`. Whereas the default behavior of `ArrayList<E>` is to add elements to the end of the list, a `LinkedList<E>` data structure can grow from **both ends** of the list.* Writing the code to implement a `LinkedList<E>` data structure is a common task that students in rigorous computer science programs often undertake, but doing so is beyond the scope of this text. A linked list data structure works by maintaining references between objects in the collection. Changing reference variables is much more memory efficient than creating new objects and copying values (as takes place in `ArrayList<E>` as it resizes). Luckily for us, Java's developers have implemented a highly efficient and easy-to-use `LinkedList<E>` class! You can use some of the more commonly used methods of the `LinkedList<E>` class like so:

Code Snippet CC5.7

```
LinkedList<String> daysOfWeek = new LinkedList<>();

daysOfWeek.add("Wednesday");
daysOfWeek.addFirst("Monday");
daysOfWeek.addLast("Friday");
daysOfWeek.add(1, "Tuesday");
daysOfWeek.add(3, "Thursday");

daysOfWeek.addLast("Saturday");

System.out.println(daysOfWeek.getLast() + "\n");
daysOfWeek.removeLast();

while (!daysOfWeek.isEmpty())
{
    System.out.println(daysOfWeek.getFirst());
    daysOfWeek.removeFirst();
}
```

Running this code produces the following output:

```
Saturday

Monday
Tuesday
Wednesday
Thursday
Friday
```

The `LinkedList<E>` class also provides some convenient search and search-based removal methods as well:

Code Snippet CC5.8

```
LinkedList<Integer> numberList = new LinkedList<>();

numberList.add(15);
numberList.add(21);
numberList.add(10);
```

* https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/LinkedList.html.

```
numberList.add(21);
numberList.add(10);
numberList.add(21);
numberList.add(7);

// Print the initial list
printCollection(numberList);

// Print index of first occurrence of 21
System.out.println(numberList.indexOf(21));

// Print index of last occurrence of 21
System.out.println(numberList.lastIndexOf(21));

// Remove the last occurrence of the duplicated 21
System.out.println(numberList.removeLastOccurrence(21));

// Remove the first occurrence of the duplicated 21
System.out.println(numberList.removeFirstOccurrence(21));

// Print the resulting LinkedList after modifications
printCollection(numberList);
```

This will result in the following output when executed (with author notes added):

```
15 21 10 21 10 21 7          (Initial LinkedList)
1                            (Index of first 21)
5                            (Index of last 21)
true                         (Successfully removed last 21)
true                         (Successfully removed first 21)
15 10 21 10 7                (Final state of the LinkedList)
```

The `LinkedList<E>` data structure has the efficiency advantage over `ArrayList<E>` in adding elements to the front/start of the collection. This efficiency can be seen when working with very large collections of elements in complex information systems. Adding an element at the beginning of an `ArrayList<E>` data structure would incur lots of memory usage and object copying. `LinkedList` data structures are more efficient due to the use of reference variables to build the "chain" of elements in the collection. For both data structure types, the use of their index notation ability is not recommended, as it is less efficient than using the built-in methods like `.addFirst()` or `.removeFirst()`. When printing the contents of these collections, the use of the enhance `for` loop (`for-each` loop) is also recommended, as it will take advantage of each class's internal implementation of their data handling. Using a standard `for` loop and index notation with these will work but grows more inefficient (i.e., slower) with larger sizes of collections.

SUMMARY POINTS

- A collection is a group of data usually implemented in programming languages in objects called data structures.
- The Java `Collections` framework is a family of interfaces, abstract classes, and concrete classes that provide Java developers with convenient functionality for quickly and effectively integrating collection data structures into their applications.
- The classes of the `Collections` framework are generically typed using type parameters, allowing flexibility as to the data types collection objects can store and manage.
- `Collections` classes cannot store primitive data types but can store objects of the wrapper class equivalents (`Integer`, `Double`, `Float`, etc.), which are subclasses of the Java class Number.
- The `ArrayList<E>` class is a commonly used collections class, useful in that it implements a resizable array. `ArrayList<E>` replaces the older `Vector<E>` class as the preferred resizable array in Java.

- List and Queue objects allow duplicate elements to be added to their collections, whereas Set and Map objects do not, since the elements themselves serve as the "index" position values.
- The Collections class differs from the root-level interface Collection<E>. The former provides the developer with many convenient methods for working with and manipulating objects of the Collections framework classes.
- A LinkedList<E> object is highly efficient at adding and removing items from the front and back of a collection due to its use of reference variables as the "links" in the list.

QUICK PROBLEMS

1. **Coding:** Write a small program that will fill two separate ArrayList<E> collections, each with one hundred randomly generated whole (int) numbers. Call the appropriate method to delete all numbers from one ArrayList<E> that are found in the other. Run this program three to four times, and print the resulting size of the altered ArrayList<E>. On average, how many numbers are removed?
2. **Think:** Consider how you might implement your own version of an ArrayList<E> class using a normal array. What types of memory inefficiencies may occur because of the use of a normal array?
3. **Coding:** Create a small ArrayList<E> collection where each element in the collection is also itself an ArrayList<E>. Add a few numbers to each of these, and use a nested for-each loop to print the contents to the console.

CC5.2 Stacks, Queues, Sets, and Iterators

Figure CC5.4. Stack of Pancakes
Source: "Mount Pancake" by smittenkittenorig, https://www.flickr.com/photos/83202873@N00/4451172538.

The ArrayList<E> data structure in Java is the "closest" in its nature and behavior to a standard array, so it makes for a great first stop in your understanding of the Collections framework classes. As you learned earlier in the chapter, ArrayList<E> and LinkedList<E> class objects offer a much richer set of functionality and behavior than a standard array—namely, dynamic resizing and searching/sorting/manipulation methods that do a lot of the data handling work for you. Moving beyond these into the Collections framework, data structures like stacks, queues, and sets offer the same collection of data in one object with processing methods. Additionally, they overlay natural default behaviors that make them useful in various contexts.

Working with a stack: The Vector<E> class implements the List<E> interface in the Java Collections framework.* Vector is an older class in the Java API and was, originally, the "go-to" class when developers needed a dynamically resizable array (prior to ArrayList<E>). One major benefit is that the Vector<E> class is synchronized, making it useful in multithreaded, parallel processing situations. Otherwise, the use of ArrayList<E> is recommended by the Oracle Java documentation when thread safety is not immediately needed.

The Stack<E> class inherits from Vector<E> and adds to its functionality the management of its elements in a last in, first out (LIFO) manner.† Think of a stack of pancakes (i.e., "hot cakes," pictured in figure CC5.4). If you place them on a plate one at a time, one on top of another, you have formed a stack. You cannot take the first one off the plate without taking the others above it off first. So the last pancake placed on top would be the first to come off. Drawing from a deck of cards might work the same way in a card game. Other real-life examples of stacks are a line of automobiles in a driveway, a stack of printer paper inside a printer tray, a stack of potato crisps in a potato chip can, and so on. In many of these situations, you are (usually) forced to remove them starting with the one on top. Only rebels pour out the entire contents of the potato chip can!

* https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Vector.html.
† https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Stack.html.

Since a Stack<E>-based object manages its collection in a LIFO manner, it provides the following methods for working with data in its collection:

- .push(E object): Push a new value onto the top of the stack.
- .peek(): Return a copy of the topmost item on the stack, but do not remove it.
- .pop(): Return a copy of the topmost item on the stack and remove it.
- .search(Object obj): Search the stack for a match to the parameter object. This method will return back an int value that is the **distance from the top of the stack** where the object is located. The topmost item is at distance 1, the next 2, and so on.

You can see the usage of a Stack<E> object easily. Note that the printCollection() method defined and implemented earlier in this chapter is used in this and future examples:

Code Snippet CC5.9

```
Stack<String> cities = new Stack<>();
cities.push("Baltimore");
cities.push("Linden");
cities.push("Martin");
cities.push("Auburn");

System.out.println(cities.peek());
printCollection(cities); // Author's collection print method
System.out.println(cities.search("Auburn"));
System.out.println(cities.search("Linden"));
System.out.println(cities.pop());
System.out.println(cities.pop());
printCollection(cities);
```

When run, this will produce the following console output (with author's notes added):

```
Auburn                          (.peek() at topmost item)
Baltimore Linden Martin Auburn  (print the list using author's method)
1                               ("Auburn" is topmost)
3                               ("Linden" is in 3rd position)
Auburn                          (Pop the topmost item off)
Martin                          (Pop the topmost item off)
Baltimore Linden                (Contents of list after edits)
```

Since the Stack<E> class inherits from Vector<E>, all the methods from Vector<E> are available for use, like the overloaded .add() method for adding an element at a particular index position in the stack, .remove() for removing from a position other than the top, .setSize() to resize the stack and discard elements if you shrink its size, and so on. Though these are available, its advised to not use them with a Stack<E> (or if you need them, consider an ArrayList<E> or other Collections class type), since they detriment the benefits of using a Stack<E> in the first place!

Figure CC5.5. A Queue of Birds
Source: "The Queue (CC)" by marfis75 Martin Fisch is licensed under CC BY 2.0, https://www.flickr.com/photos/45409431@N00/4289431717.

Working with queues: When you picture a queue (pronounced like the letter "q"), think of a line of people (or birds, as in figure CC5.5). In fact, in Great Britain and other European countries, a line of people is normally called a "queue," or for the verb form, you would say that people have "queued up." Whether or not the Western Hemisphere should adopt this terminology is beyond the scope of this textbook. For our purposes, the Java collections framework includes the Queue<E> interface to allow data structures to work like a line of people: the first person in is usually the first person out, or FIFO ("first in, first out"). Elements in a queue data structure are added to the end and are removed from the beginning (the "head").

Queues in Java are generally created using the LinkedList<E> class, as it is highly efficient at adding and removing elements from both ends of its collection.

LinkedList<E> implements the Deque<E> interface (pronounced "deck"), which is a subinterface of Queue<E>. In addition to other methods (since LinkedList<E> also implements the List<E> interface), the following Queue<E> methods are available for manipulating the data in the collection:

- .offer(E element): This will add an element to the head of the queue ("front of the line"). It will return a true or false based on the success of this addition (false if there are capacity issues).
- .peek(): Returns a copy of the element at the head of the queue but does not remove it. Will return null if the queue is empty.
- .poll(): Returns a copy of the element at the head of the queue and removes it. Will return null if the queue is empty.
- .element(): Works the same as .peek() except that it will throw a NoSuchElement exception if the queue is empty instead of a null value.
- .remove(): Works the same as .poll() except that it will throw a NoSuchElement exception if the queue is empty instead of a null value.

Like with a stack, a queue data structure is fairly simple to demonstrate in our Java main() application:

Code Snippet CC5.10

```
Queue<String> bankLine = new LinkedList<>();

bankLine.offer("Mary");
bankLine.offer("Sarah");
bankLine.offer("Aayliah");
bankLine.offer("Arquette");

printCollection(bankLine); // Author's collection print method

System.out.println("Customer: " + bankLine.poll() + " finished transaction.");
System.out.println("Customer: " + bankLine.peek() + " is next.");
System.out.println("Customer: " + bankLine.poll() + " finished transaction.");
System.out.println("Customer: " + bankLine.peek() + " is next.");

printCollection(bankLine); // Author's collection print method
```

When run, this will produce the following console output (with author's notes in **bold**):

```
Mary Sarah Aayliah Arquette                 (Initial line)
Customer: Mary finished transaction.        (Customer leaves)
Customer: Sarah is next.
Customer: Sarah finished transaction.        (Customer leaves)
Customer: Aayliah is next.
Aayliah Arquette                             (Line after processing)
```

Stacks, queues, and efficiency in Java: The Oracle documentation for the Stack<E> class and the tutorial documentation for the Deque<E> interface recommend the use of Deque<E>, which the ArrayDeque<E> class implements, over the use of a Stack<E>.[*,†] The documentation mentions that for a regular queue, the ArrayDeque<E> collection has better efficiency and speed in various tasks over the use of a LinkedList<E> as well. The LinkedList<E> class is useful though: it has implemented more functionality of the List<E> interface, and null elements are allowed, whereas they are not allowed in an ArrayDeque<E>.

ArrayDeque<E> collections, like LinkedList<E> objects, are highly efficient at adding and removing elements at both ends of the list. This makes them efficient when used as either a stack or a queue in Java. ArrayDeque<E> has methods implemented for both uses:

[*] https://docs.oracle.com/javase/tutorial/collections/implementations/deque.html.

[†] https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/ArrayDeque.html.

Code Snippet CC5.11

```
Deque<String> customersWaiting = new ArrayDeque<>();
Deque<Integer> containerIDStack = new ArrayDeque<>();

// Using customersWaiting as a Queue

customersWaiting.offer("Aliyah");
customersWaiting.offer("Markus");
customersWaiting.offer("Salle");

printCollection(customersWaiting);
System.out.println("Now Serving Customer: " + customersWaiting.peek());
System.out.println(customersWaiting.poll());
System.out.println("Now Serving Customer: " + customersWaiting.peek());
System.out.println(customersWaiting.poll() + "\n\n");

// Using containerIDStack as a Stack

containerIDStack.push(45);
containerIDStack.push(65);
containerIDStack.push(12);

printCollection(containerIDStack);
System.out.println("Retrieving Container #: " + containerIDStack.peek());
System.out.println(containerIDStack.pop());
System.out.println("Retrieving Container #: " + containerIDStack.peek());
System.out.println(containerIDStack.pop());
```

This, when run, will produce the following output:

```
Aliyah Markus Salle
Now Serving Customer: Aliyah
Aliyah
Now Serving Customer: Markus
Markus

12 65 45
Retrieving Container #: 12
12
Retrieving Container #: 65
65
```

With such small collection sizes as these, the efficiency gains are invisible, but the use of `LinkedList<E>` and `ArrayDeque<E>` are available if needed as your application scales larger.

Working with a `PriorityQueue<E>`: A relative of the `Queue<E>` implementing classes in Java is the `PriorityQueue<E>` class. Data structures of this class will perform FIFO actions on the elements in this collection not based on the order they were inserted but rather based on their natural order. For example, consider the following code:

Code Snippet CC5.12

```
PriorityQueue<Integer> orderedQueue = new PriorityQueue<>();

orderedQueue.offer(34);
orderedQueue.offer(15);
orderedQueue.offer(3);
orderedQueue.offer(64);
orderedQueue.offer(6);

while (orderedQueue.size() >= 1)
{
    System.out.println(orderedQueue.poll());
}
```

In a normal queue as exampled earlier, the first element to be removed with the invocation of `.poll()` on the collection would be the number `34`, since it was the first in. But with a `PriorityQueue<E>` collection, here the "head" of the line is the integer with the lowest value. Behind that is the next largest integer, and so on. The console output for this example code would be the following:

```
3
6
15
34
64
```

For class objects, as long as the class has implemented the `Comparable` interface (see companion chapter 4), then a "natural ordering" can be determined and its objects used with a `PriorityQueue<E>` collection.

Working with `Set<E>` collections: The collection classes that implement the `Set<E>` interface have adopted behaviors that demonstrate a strict control over the elements placed within their collection objects. Some things to keep in mind with a `Set` collection in Java:

- A `Set` collection cannot contain duplicate values. If you attempt to add a duplicate value into a `Set` collection, the addition will fail, and usually a false boolean value will be returned from the `.add` method.
- `Set` collections do not use index notation at all in their mapping of elements. The value of an element in a `Set` in Java serves as its index. Various classes that implement `Set<E>` will use the value of the element itself in various ways to order the elements in the collection.

The **`HashSet<E>`** class implements the `Set<E>` interface. Since a `set` in Java does not use index notation, elements in the collection for a `HashSet<E>` are arranged by creating a hash code based on the element's value, and elements are ordered by that hash code. From the developer side, we do not have access to the hashed value of the elements we add, and there is no specific order that a `HashSet<E>` will impose on our elements. The `HashSet<E>` will impose an order that will maximize the speed at which you can test for a duplicate value in the `HashSet<E>` and remove elements, among other actions. In many information systems applications, in-memory sets of data can grow very large, and ordering the values by a hash code system can enable fast read/write even as the set grows larger. You can explore the usage of a `HashSet<E>` collection like so:

Code Snippet CC5.13

```
Set<Integer> hashedSet = new HashSet<>();
hashedSet.add(34);
hashedSet.add(20);
hashedSet.add(134);
hashedSet.add(134);
hashedSet.add(2);

printCollection(hashedSet);
```

This will produce the following console output:

```
34 2 20 134
```

Notice that the code attempts to add the value `134` twice, but the value was only inserted once. In a set, duplicates are not allowed (due to the fact that the value itself serves as its own index notation). The `.add()` method returns `false` when a duplicate is found. Since the `HashSet<E>` class also implements the `Collection<E>` interface and provides concrete implementation of its methods, you can use a `HashSet` for more traditional logical set operations:

Code Snippet CC5.14

```
HashSet<String> citiesNeedingUpdates = new HashSet<>();
citiesNeedingUpdates.add("Milan");
citiesNeedingUpdates.add("Jackson");
citiesNeedingUpdates.add("Lexington");
citiesNeedingUpdates.add("Trezvant");
citiesNeedingUpdates.add("Murfreesboro");
```

```
citiesNeedingUpdates.add("Antioch");

Set<String> updatedRoads = new HashSet<>();
updatedRoads.add("Antioch");
updatedRoads.add("Jackson");
updatedRoads.add("Milan");

citiesNeedingUpdates.removeAll(updatedRoads);

printCollection(citiesNeedingUpdates); // Author's Print Method
```

This will print the following to the console:

```
Murfreesboro Trezvant Lexington
```

Traditional operations possible with a `Set<E>` collection are the following:

- **Set difference:** Use the `.removeAll()` method as exampled here.
- **Union:** Use the `.addAll()` method (remember the duplicates will be skipped!).
- **Intersection:** Use the `.retainAll()` method.

The **`TreeSet<E>`** class uses not hashing but the more familiar natural ordering of elements to arrange elements within the collection. Elements are stored in a treelike fashion (node and two “leaves”) in ascending order, and like most sets, this class does not allow for duplicates. Consider the following code:

Code Snippet CC5.15

```
TreeSet<Integer> binaryTree = new TreeSet<>();
binaryTree.add(45);
binaryTree.add(11);
binaryTree.add(3);
binaryTree.add(15);
binaryTree.add(46);
binaryTree.add(90);
binaryTree.add(87);

printCollection(binaryTree); // Author's print method
```

This will print the following to the console:

```
3 11 15 45 46 87 90
```

Starting with the first number, if the next number is less than that value, it will be stored in a left branch, and numbers larger in the right. Figure CC5.5 shows how our values are associated in a treelike structure after addition to the `TreeSet<E>` collection.

Some interesting methods implemented for the `TreeSet<E>` class to be aware of are the following:

- `.first()`: Will return a copy of the first element from the `TreeSet<E>` collection based on natural ordering.
- `.last()`: Will return a copy of the last element from the collection based on natural ordering.
- `.higher(E element)`: Will take the element specified in the parameter and return the next highest element from the `TreeSet<E>`. For example, if `34` and `35` are in the list, and `34` is the parameter, then `35` would be returned, as it is the next highest element.
- `.lower(E element)`: Will take the element specified in the parameter and return the next lowest element from the collection.
- `.pollFirst()`: Will return a copy of and remove the first element in the collection based on natural ordering.
- `.pollLast()`: Will return a copy of and remove the last element in the collection based on natural ordering.

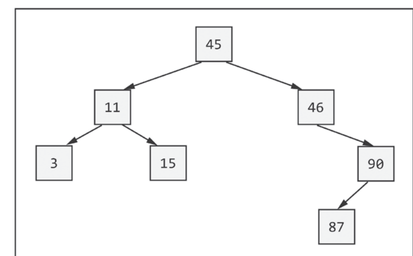

Figure CC5.6. Binary Search Tree of Values Added to `binaryTree` Collection

For example, you can remove the largest numbers from our `binaryTree` `TreeSet<E>` collection by calling `.pollLast()` while the set is not empty:

Code Snippet CC5.16

```
while (binaryTree.size() > 0)
{
    System.out.println("Removing: " + binaryTree.pollLast());
    printCollection(binaryTree); // Author's print method.
}
System.out.println("TreeSet is empty!");
```

This prints the following to the console:

```
Removing: 90
3 11 15 45 46 87
Removing: 87
3 11 15 45 46
Removing: 46
3 11 15 45
Removing: 45
3 11 15
Removing: 15
3 11
Removing: 11
3
Removing: 3
TreeSet is empty!
```

`TreeSet<E>` has the advantage over a `HashSet<E>` in that elements are naturally ordered (for complex data types, the `Comparable` interface provides the implementation in these classes for determining "value"). For speed and efficiency, `HashSet<E>` is typically faster for addition, removal, and search operations in those sets that grow very large.

Working with iterators: Each of the classes in the Java `Collections` framework implements its collection data operations in its own unique way, creating a unique structure internally that is normally hidden from the developer. Additionally, each class implements its own set of methods for interacting with the data stored in the internal collection. For example, we `.add()` to a LinkedList, `.offer()` to a queue, and `.push()` to a stack. The way to retrieve values from these collections can be just as unique. Luckily for us, the developers of Java have carried the handy concept of abstraction into the `Collections` framework with the inclusion of the `Iterable<T>` interface. You have heard the term "to iterate" before in our discussion of loops (see chapter 5). The `Iterable<T>` interface includes one abstract method, `.iterator()`. Classes that implement `Iterable<T>` do so with the following intentions:

- Providing a common, simple way of traversing (i.e., walking through) all the elements of a collection at once
- Providing read-only access to the elements
- Providing an algorithmically efficient (i.e., fast!) way of traversing the elements in a collection
- Providing an abstracted interface using methods you are already familiar with

Classes that implement the `Iterable<T>` interface provide a concrete implementation of `.iterator()` (and sometimes more). The `.iterator()` method creates and returns a reference to an `Iterator<E>` object, which performs the walk across a data structure. For example, with our `TreeSet<E>` collection from earlier, we can use an iterator to walk through the elements and print them in an efficient way:

Code Snippet CC5.17

```
TreeSet<Integer> binaryTree = new TreeSet<>();
binaryTree.add(45);
binaryTree.add(11);
binaryTree.add(3);
binaryTree.add(15);
binaryTree.add(46);
binaryTree.add(90);
```

```
binaryTree.add(87);

Iterator<Integer> treeWalk = binaryTree.iterator();
while (treeWalk.hasNext())
    System.out.print(treeWalk.next() + " ");

System.out.println();

Iterator<Integer> treeBackWalk = binaryTree.descendingIterator();
while (treeBackWalk.hasNext())
    System.out.print(treeBackWalk.next() + " ");
```

This will print to the console the following:

```
3 11 15 45 46 87 90
90 87 46 45 15 11 3
```

Notice that `Iterator<E>` objects use `.hasNext()` and `.next()`, the same methods you are familiar with from the `Scanner` class. Additionally, the use of an `Iterator<E>` object simplifies collection traversal. You do not have to know how the class implements its internal structure or the methods used to walk through it, as it abstracts and simplifies all that through the `Iterator<E>` object it provides. The `TreeSet<E>` class implements an additional method, `.descendingIterator()`, which provides a reverse-walking iterator object to move through the natural order of the `TreeSet<E>` elements in reverse. The `LinkedList<E>` class does something similar as you can see here:

Code Snippet CC5.18

```
LinkedList<String> userNames = new LinkedList<>();
userNames.add("markujm");
userNames.add("samulsi");
userNames.add("epirzx");
userNames.add("andermm");
userNames.add("deverbr");

Collections.sort(userNames);

ListIterator<String> usernamesIter = userNames.listIterator(userNames.size());
while (usernamesIter.hasPrevious())
    System.out.print(usernamesIter.previous() + " ");
```

Console output:

```
samulsi markujm epirzx deverbr andermm
```

Using an enhanced `for` loop (`for-each` loop) is a similarly fast and efficient way of traversing the elements in a collection as well. For convenience, the definition for our custom `printCollection()` method is shown here:

Code Snippet CC5.19

```
public static void printCollection(Collection<?> dataStructure)
{
    for (Object obj: dataStructure)
    {
        System.out.print(obj + " ");
    }
    System.out.println();
}
```

Oracle's documentation recommends the use of either an enhanced `for` loop or, preferably, an `Iterator<E>` to traverse the elements in a collection in an efficient way.

SUMMARY POINTS

- A `Stack<E>` collection inserts items at the end of the collection and removes them from there as well, implementing a last in, first out (LIFO) data-handling approach.
- A `Queue<E>` collection inserts items at the end of the collection but removes them from the head, implementing a first in, first out (FIFO) data-handling approach.
- The Oracle Java documentation recommends using an `ArrayDeque<E>` in both Stack and Queue situations, as it is efficient at element manipulation at both ends of a collection.
- The `PriorityQueue<E>` will operate in a LIFO manner but order elements for removal based on their natural ordering (via the `Comparable<E>` interface).
- In a set, the value of the element itself serves as the means by which the collection of elements is ordered. Sets do not use index notation, as the element's value itself serves as the "index position" of where it belongs in the collection.
- `HashSet<E>` collections create a hash code of the element's value and order the collection via these hashes. `HashSet<E>` is extremely efficient at adding/searching/removing operations.
- `TreeSet<E>` will order elements according to their natural order/values.
- Iterators provide a convenient way for developers to interact with all objects created with the `Collections` framework in a standard and familiar way, regardless of the implementation and interaction details unique to each class.

QUICK PROBLEMS

1. **Coding:** Implement a small program that loops in a menu system that has four options presented to the user: "Jump," "Dance," "Sing," "Laugh." Loop so that the program lets the user choose any one they want during each loop, and limit them to ten choices. Save their choices in a queue, and print their history out at the end of the program.
2. **Think:** How can the usage of `Iterable<T>` allow for interoperability across objects instantiated with the `Collections` classes?
3. **Coding:** Generate ten random integers (whole numbers), and place five of each into two separate stacks (use `Stack<E>` or `ArrayDeque<E>`). Write some logic that will loop through both stacks and compare the topmost numbers from each. Print the smallest of the two, remove it from its stack, and perform the comparison again with the current top two numbers from each stack. Repeat until both stacks are empty.

CC5.3 Using the Map Classes of the `Collections` Framework

The "second half" of the Java `Collections` framework deals with the `Map<K,V>` interface and the classes that provide concrete implementations of it and its subinterfaces. Instead of a "list" of single elements, each element in a map collection stores two pieces of data: a **key** and a **value** that is mapped to that key. An analogy here would be to imagine a normal array where the index position is stored as an `int` alongside the "chunk" of data in the array at each element. Figure CC5.8 shows an example of a mapping (sometimes called a dictionary or associative array in the field of computer science).

In the Java `Collections` framework, `Map<K,V>` objects have the following characteristics:

- Maps in Java cannot contain duplicate values. Map classes use the key as the "index" position and order the key-value pairs using the key, preventing duplicates.

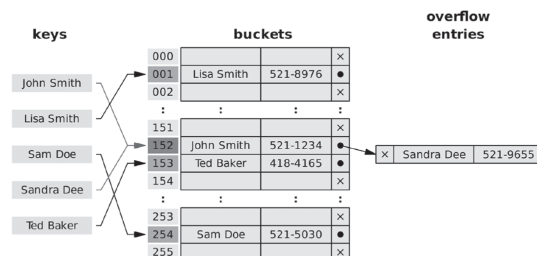

Figure CC5.7. Common Example of a `Map` Structure with Multiple Values per Key
Source: "Hash Table Illustration" by Jorge Stolfi is licensed under CC BY-SA 3.0, https://commons.wikimedia.org/wiki/File:Hash_table_5_0_1_1_1_1_0_LL.svg.

- The classes implementing the `Map<K,V>` interface in Java **do not** implement the `Iterable<T>` interface.
- The `HashMap<K,V>` class produces a hash code of the key's value (not the mapped value) to organize the map entries, making this very fast at both adding (and testing for duplicates) and retrieving and removing as your list of entries grows very large.
- The `TreeMap<K,V>` class produces a Map collection that is naturally ordered/sorted by the key. Useful when you need an ordered key/value collection.
- The `LinkedHashMap<K,V>` class allows the developer to preserve the order of insertion into the map while preserving the speed benefits of both a linked list and hashed keys for addition/retrieval/removal as the collection grows large.

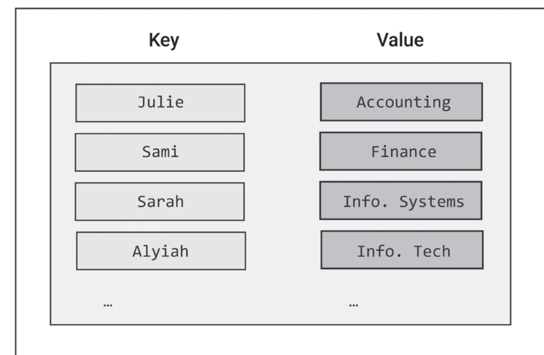

Figure CC5.8. Example Map of Key and Value Pairs

Table CC5.4 briefly describes some of the more commonly used methods of the `Map<K,V>` interface and its implementing classes:

Table CC5.4. Descriptions of Select Methods of the Map<K,V> Interface

Map<K,V> method	Description
`.put(K key, V value)`	Places a new entry into the Map collection, with `value` mapped to `key`.
`.get(Object key)`	Returns a copy of the `value` mapped to `key`.
`.remove(Object key)`	Returns a copy of the `value` mapped to the `key` in the `Map` and removes the `key/value` mapping from the collection.
`.replace(K key, V value)`	Replaces the `value` mapped to `key` if the mapping exists in the collection.
`.containsKey(Object key)`	Returns a `boolean true/false` testing if the map contains the parameter `key`.
`.containsValue(Object value)`	Returns a `boolean true/false` testing if the map contains the parameter `value`.
`.keySet()`	Returns a Set<K> object that contains all the keys from the map.
`.values()`	Returns a Collection<V> object that contains all the values from the map.
`.entrySet()`	Returns a Set collection whose elements are data typed to Map.Entry<K,V> objects, essentially returning a set of the mapping pairs.
`.forEach(BiConsumer<? super K, ? super V> action)`	Used instead of an `Iterable<T>` implementation. Accepts as a parameter a reference to an instance object whose class has implemented the `BiConsumer<…>` interface.* Often used with a lambda expression.

* https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/function/BiConsumer.html

An example usage of the `HashMap<K,V>` class would be as follows:

Code Snippet CC5.20

```
Map<String, String> employeeAddresses = new HashMap<>();

employeeAddresses.put("Market Street", "Jamie K.");
employeeAddresses.put("Andover Lane", "Margie L.");
employeeAddresses.put("Terrapin St.", "Jerry G.");

// 1. Traversing the Map using the KeySet and .get()
for (String add : employeeAddresses.keySet())
{
```

```
    System.out.println(
            "Employee: " + employeeAddresses.get(add)
                    + ", Address: " + add);
}

// 2. Traversing the Map using the EntrySet and key/value in each.
for (Map.Entry<String, String> entry : employeeAddresses.entrySet())
{
    System.out.println(
            "Employee: " + entry.getValue()
                    + ", Address: " + entry.getKey());
}

// 3. Traversing the Map using a lambda expression
employeeAddresses
        .forEach((address, name) -> System.out.println(
                                "Employee: " + name
                                        + ", Address: " + address));
```

Notice that included in this example are three separate ways to traverse the `HashMap<K,V>` and print out its values:

1. By invoking the `.keySet()` method to retrieve a `Set<K>` of keys, then calling `.get()` to extract the value mapped to each key.
2. By invoking the `.entrySet()` method to extract a Set with `Map.Entry<K,V>` data-type objects, then calling the `.getKey()` and `.getValue()` on each `Map.Entry<K,V>` object.
3. By using a lambda expression in the `.forEach()` method parameter. Recall from chapter 11 that lambda expressions can be used to simplify syntax. To understand this usage of `.forEach()`, keep the following in mind:
   a. The `.forEach()` method accepts as a parameter a reference to a class that implements the `BiConsumer<? super K, ? super V>` interface. This interface defines the abstract method `.accept(T t, U u)`.
   b. The lambda expression creates an anonymous inner class, has it implement the `BiConsumer<…>` method `.accept()`, and passes an object of this anonymous inner class to the `.forEach()` method—all behind the scenes!
   c. The lambda expression allows us to create custom names for the "input" parameters that come from the Map entries, which here are `address` and `name`. The code after the `->` represents the code that would have been written for the `.accept()` method from the `BiConsumer<…>` interface.

The output from all three versions of the traversal of this `HashMap<K,V>` looks the same:

```
Employee: Margie L., Address: Andover Lane
Employee: Jamie K., Address: Market Street
Employee: Jerry G., Address: Terrapin St.
```

Note as well that the order the entries are printed is not the same as inserted. Recall that a hash code is generated for the key internally in the `Map<K,V>` structure (similar to the behavior of `HashSet<E>`) as it manages its own internal order. So print and retrieval order may not match insertion for a `HashMap<K,V>`.

Business system example using HashMap<K,V>: A common usage and example of a `HashMap<K,V>` would be to perform a count of an unknown (ahead of time) set of key values. For example, in an information system with many users, you cannot predict the order or the frequency that logins will occur by those users. A `HashMap<K,V>` is handy in that if a new key (here a username) is encountered and is not already mapped, it will be added. Consider the following example:

Code Snippet CC5.21

```
// History of user logins today
String[] users = {"A","C","A","D","E","F","D","C","C","F","M","A"};
HashMap<String, Integer> loginCount = new HashMap<>();
int tempCount = 0;

for (String c : users)
{
    if (!loginCount.containsKey(c))
        loginCount.put(c, 1);
    else
    {
        tempCount = loginCount.get(c);
        tempCount++;
        loginCount.replace(c, tempCount);
    }
}

loginCount.forEach((user, count) -> System.out.println(
        "Employee: " + user + ", Login Count: " + count));
```

User logins are mimicked here by using an array of `String` values that represent usernames. Notice that this code first checks to see if the next login has been counted previously. If not, it is added to the `Map<K,V>` with the username as a key and the count, `1`, as the value mapped to the key. If the login has occurred previously, then the prior count is retrieved, incremented, and replaced in the mapping. The output from this code execution would be the following:

```
Employee: A, Login Count: 3
Employee: C, Login Count: 3
Employee: D, Login Count: 2
Employee: E, Login Count: 1
Employee: F, Login Count: 2
Employee: M, Login Count: 1
```

This is a great example of how useful `Map<K,V>` structures can be. The collection itself handles the searching, matching, and mapping without the need for a complicated `switch` statement. Since you cannot guess ahead of time which users might log in, a hard-coded `switch` would not work well.

SUMMARY POINTS

- A map data structure in a program stores pairs of keys and values. Each value is mapped to one key, and duplicates are not allowed.
- The `HashMap<K,V>` class orders the key/value pairs by producing a hash code of the key. `HashMap<K,V>` is highly efficient at add/search/remove operations.
- The `TreeMap<K,V>` class arranges key/value pairs by the natural ordering of the keys.
- The `LinkedHashMap<K,V>` uses a hash code of the key in a key/value pair to locate mappings in the collection but will also preserve the insertion order of key/value pairs.
- Classes implementing the `Map<K,V>` interface implement the `.forEach()` method, which can be used with a lambda expression to quickly traverse the map. A regular enhanced `for` loop can also be used to efficiently traverse a map collection.

QUICK PROBLEMS

1. **Coding:** Add five student names and their year in college (freshman, sophomore, junior, senior) to a `TreeMap<K,V>` collection. Think about which should be the key and which the value. Afterward, print out from the map only those students who are "junior" in year classification.

2. **Think:** When might preserving a custom ordering via insertion using a `LinkedHashMap<K,V>` be preferable to allowing the map collection to order the elements itself (as in `TreeMap<K,V>` or `HashMap<K,V>`)?
3. **Coding:** Create a `HashMap<K,V>` to store telephone prefixes and the cities that fall under each. For the value, use an `ArrayList<String>`. Add three prefixes and two to four cities to each.

CC5.4 Additional Business Applications of Some Java `Collections` Framework Classes

In this section, brief examples are given where several of the major classes of the Java `Collections` framework can be used in business application scenarios. Code examples are provided for most. Provided code examples can be implemented within the `main()` method of a Java class.

`ArrayList<E>`: The `ArrayList<E>` class provides a flexible, dynamically resizable array along with several powerful data manipulation methods. Whenever a list of data needs to be stored and you are unsure of **how many** elements will be used, an `ArrayList<E>` is a good choice. For example, a simple shopping cart can be implemented where a user could buy any number of items from a grocery store:

Full Program CC5.1.A

```
import java.util.*;

public class CollectionsExamples {
    public static void main(String[] args) {
        Scanner keyboardIn = new Scanner(System.in);
        ArrayList<String> shoppingCart = new ArrayList<>();
        String item = "";
        while (true)
        {
            System.out.print("Enter Grocery Item: ");
            item = keyboardIn.nextLine();
            if (!item.equalsIgnoreCase("done"))
                shoppingCart.add(item);
            else
                break;
        }

        Collections.sort(shoppingCart);

        System.out.println("\nShopping Cart Contents:");
        for (var purchasedItem : shoppingCart)
            System.out.println(purchasedItem);

    } // End of main()
} // End of CollectionsExamples class
```

Running this code would yield the following output:

```
Enter Grocery Item: Eggs
Enter Grocery Item: Cheese
Enter Grocery Item: Celery
Enter Grocery Item: Salad Mix
Enter Grocery Item: Arugula
Enter Grocery Item: Kale
Enter Grocery Item: done

Shopping Cart Contents:
Arugula
Celery
Cheese
Eggs
Kale
Salad Mix
```

LinkedList<E>: The Collections class LinkedList<E> works in a similar way to ArrayList<E>, but the name of the game here is speed. A LinkedList<E> is much faster with the insertion and deletion of items ad hoc (i.e., anywhere) in the list when the amount of data in the list is very large.

Full Program CC5.1.B

```
import java.util.*;

public class CollectionsExamples {
    public static void main(String[] args) {

        LinkedList<String> itemList = new LinkedList<>();
        String[] items = {"Corn","Eggs","Fajita Mix","Kale","Mangos"};

        for (var item: items)
        {
            itemList.add(item);
        }

        itemList.add(3,"Fruit Mix"); // insert at index 3
        itemList.addFirst("Apples"); // insert at start

        for (var item: itemList)
        {
            System.out.println(item);
        }
    } // End of main()
} // End of CollectionsExamples class
```

The output when running this example would look like the following (with inserted grocery items in **bold**):

Apples
Corn
Eggs
Fajita Mix
Fruit Mix
Kale
Mangos

The usage of LinkedList<E> allows an insertion at the start, end, or anywhere in the list of data to occur in a highly efficient and speedy manner, with little regard to how much data is in the list. With an ArrayList<E>, an array is dynamically being resized (re-created and items copied behind the scenes). With a LinkedList<E>, each element is a stand-alone element, and reference variables between the elements are rearranged with insertions/deletions.

Stack<E>: A Stack<E> can be useful when a business needs to process the most recent data or events first and then move on to older data. For example, often a company will employ a last in, first out (LIFO) accounting method to process more recently created inventory as having been sold "first," since recent products often yield more revenue due to currency inflation. Consider the following code:

Full Program CC5.1.C

```
import java.util.*;

public class CollectionsExamples {
    public static void main(String[] args) {

        Stack<Product> productRevenue = new Stack<>();

        productRevenue.add(new Product(2020,5.60));
        productRevenue.add(new Product(2021,5.90));
        productRevenue.add(new Product(2022,6.10));
```

```
        productRevenue.add(new Product(2023,8.60));

        while (!productRevenue.empty())
        {
            System.out.println(productRevenue.pop());
        }

        } // End of main()
} // End of CollectionsExamples class

// inner class
class Product{
    int year;
    double cost;

    public Product(int year, double cost)
    {
        this.year = year;
        this.cost = cost;
    }

    public String toString()
    {
        return this.year + " " + this.cost;
    }
}
```

When this code is run, the following output is produced:

```
2023 8.6
2022 6.1
2021 5.9
2020 5.6
```

Whenever you need the most recently added item to be processed first without having to keep track of insertion sequence, a `Stack<E>` is the way to go.

`TreeSet<E>`: The `TreeSet<E>` class provides a way to insert items into a `Set` where items are automatically inserted in a sorted, ordered manner. This can save the programmer the time needed to write code that would manage the sorting of items themselves. Consider the shopping list example from earlier, this time modified to use a `TreeSet<E>`:

Full Program CC5.1.D

```
import java.util.*;

public class CollectionsExamples {
    public static void main(String[] args) {

        Scanner keyboardIn = new Scanner(System.in);
        TreeSet<String> shoppingCart = new TreeSet<>();
        String item = "";
        while (true)
        {
            System.out.print("Enter Grocery Item: ");
            item = keyboardIn.nextLine();
            if (!item.equalsIgnoreCase("done"))
                shoppingCart.add(item);
            else
                break;
        }
```

```
        System.out.println("\nShopping Cart Contents:");
        for (var purchasedItem : shoppingCart)
            System.out.println(purchasedItem);

    } // End of main()
} // End of CollectionsExamples class
```

This will produce the following output when run:

```
Enter Grocery Item: Zucchini
Enter Grocery Item: Grapes
Enter Grocery Item: Oranges
Enter Grocery Item: Cheese
Enter Grocery Item: Arugula
Enter Grocery Item: Cocoa Powder
Enter Grocery Item: done

Shopping Cart Contents:
Arugula
Cheese
Cocoa Powder
Grapes
Oranges
Zucchini
```

`HashSet<E>`: The `HashSet<E>` collection will create a hash code of the value of the item to be added, which allows for very rapid searching, addition, and removal operations even if the set grows very large in quantity. Speed is the biggest benefit of using this class. It will order only its internal hash codes, but it will not sort the items inserted. For example, the following is a modification of the shopping cart example:

Full Program CC5.1.E

```
import java.util.*;

public class CollectionsExamples {
    public static void main(String[] args) {

        Scanner keyboardIn = new Scanner(System.in);
        HashSet<String> shoppingCart = new HashSet<>();
        String item = "";
        while (true)
        {
            System.out.print("Enter Grocery Item: ");
            item = keyboardIn.nextLine();
            if (!item.equalsIgnoreCase("done"))
                shoppingCart.add(item);
            else
                break;
        }


        System.out.print("\nEnter Item To Search Cart: ");
        item = keyboardIn.nextLine();

        // Search for item in shopping cart
        System.out.println("\n" + item
                + " already added?: "
                + shoppingCart.contains(item));

    } // End of main()
} // End of CollectionsExamples class
```

Running this code produces the following (when searching for "Rice" in the shopping cart):

```
Enter Grocery Item: Arugula
Enter Grocery Item: Cheese
Enter Grocery Item: Cocoa Powder
Enter Grocery Item: Grapes
Enter Grocery Item: Oranges
Enter Grocery Item: Zucchini
Enter Grocery Item: done

Enter Item To Search Cart: Rice

Rice already added?: false
```

Looks like the user forgot to purchase something! A check like this can be used in several ways in a business scenario: searching for an already existing email address in a roster of thousands of employees, checking to see if an inventory item already exists by a name, inserting either a new employee email address or a new item name into a very large in-memory set of them, and so on.

Summary

In this chapter, you have explored the nature, basics, and effective usage of the classes of the Java `Collections` framework. Data structures such as resizable arrays, linked lists, stacks, queues, sets, and maps are the gold standard, go-to collection structures and approaches for highly efficient in-memory data storage and manipulation. Individuals studying in computer science programs will often learn how to write the code to implement these data structures, applying theory and relevance both from the computer science field. While a highly valuable effort (one your author encourages you to research further if you are interested!), for information systems developers, only the understanding of how these data structures work and their usage as implemented in the Java language is needed. Usage of these classes has innumerable applications in the business and information systems areas, helping solve many problems where sophisticated handling of data is needed. Often a neglected area of study, the Java `Collections` framework and knowledge of how to leverage it successfully are great tools for an information systems professional to put into their development arsenal!

Practice Problems

Terminology

Match the following terms from the chapter with their most appropriate definition:

1. Collection	a. An alphanumeric (numbers and letters) representation of a value. Used in encryption, hashing, and other cryptographic applications.
2. Data structure	b. A Java class that implements a data structure that is highly efficient at adding and removing elements from both ends of a collection in memory.
3. Interface	c. A Java class whose objects are generated by classes that implement the `Iterable<T>` interface and used to provide a common approach to the interaction of all `Collections` framework objects.
4. Abstract class	d. A Java class that implements a set-based data structure where cryptographic values for single elements are generated and used to order the elements in the collection.
5. `Collection<E>`	e. A Java interface that classes in the `Collections` framework implement in order to create collections of single elements that prevent duplication of values in the collection.

6. Collections	f. A Java class that implements a data structure that strictly controls the removal behavior of elements by following a LIFO approach.
7. ArrayList<E>	g. A Java class that orders key/value pairs in a data structure and preserves the insertion order of the pairs.
8. .removeAll(…)	h. A collection of single elements whose values and natural ordering determine their organization and removal behavior.
9. .retainAll(…)	i. A Java interface that classes in the Collections framework implement in order to create data structures that store elements as key and value mapping pairs.
10. LIFO	j. Java class that has some implemented methods but some that are not, allowing future subclasses to inherit and implement them fully and in a manner custom and beneficial to them.
11. FIFO	k. A Java class that orders key/value pairs in a data structure based on the natural value ordering of the pair's key.
12. Hash code	l. A Java class that implements a data structure that strictly controls the removal behavior of elements by following a FIFO approach.
13. LinkedList<E>	m. A Java class that implements a queue data structure where the natural ordering by value of the single elements in the collection determines their removal behavior.
14. Stack<E>	n. A Java class popularly used to create a resizable array. Replaces the older Vector<E> class in use.
15. Queue<E>	o. Approach to element removal in a collection where the most recently added element will be the first to be removed.
16. Set<E>	p. Method defined in the Collection<E> interface and implemented in Collections framework classes that removes all elements in one collection that are also found in the parameter collection.
17. ArrayDeque<E>	q. A sequence of similar-in-nature data gathered together in one object or structure.
18. TreeSet<E>	r. A Java class that orders key/value pairs in a data structure based on a cryptographic representation of the pair's key.
19. PriorityQueue<E>	s. A Java class that implements a data structure that is highly efficient for use in both stack and queue scenarios, as recommended by the Java documentation.
20. HashSet<E>	t. Approach to element removal in a collection where the earliest element added to the collection will be first to be removed.
21. Iterator<E>	u. Java interface that serves as the root class for all classes in the Java Collections framework.
22. Map<K,V>	v. Method defined in the Collection<E> interface and implemented in Collections framework classes that retains all elements in one collection that are also found in the parameter collection, discarding the rest.
23. HashMap<K,V>	w. In a programming context, an object that maintains data in a sophisticated manner internally and includes functionality for the manipulation of that data.
24. TreeMap<K,V>	x. A Java "utility" class that provides many static methods developers can find useful when working with data structures created using the Java Collections framework classes.
25. LinkedHashMap<K,V>	y. A Java class with no implemented methods and only constant data fields. Useful to allow future classes to provide concrete code implementations of their methods. Enables nonrelated classes to share commonalities.

Find the Error

In each of the following problems, carefully examine the code given, and determine the error(s)/issue(s) with each. Keep in mind, the error(s) could be syntax (code) or logic (intended outcome) based or both!

1.

```
// Assume all needed imports have been made
Scanner in = new Scanner(keyboardIn);
Stack<E> myStack = new Stack(<>);
while (myStack.hasNext())
{
    System.out.println(Enter a name:");"
    myStack.add(in.nextLine());
}
```

2.

```
TreeSet<int> orderNums = new Set<E>();
orderNums.offer(10);
orderNums.offer(20);
orderNums.offer(30);
orderNums.offer(10);
orderNums.offer(15);
orderNums.offer(25);
```

3.

```
LinkedList<Integer> linkL = new LinkedList<>();
for (int i=0; i<10; i++)
{
    linkL.add((int)(1 + Math.random()) * 11);
}
System.out.println(Collections.binarySearch(linkL, 4));
```

4.

```
// Logic to represent automobiles in a one-exit driveway
Queue<String> driveWay = new ArrayDeque<>();
driveWay.push("CarMake, Red, 23434");
driveWay.push("AutoBuilder, Green, 33321");
driveWay.push("SpeedGo, Polkadot, 33400");
driveWay.push("CarMake, Yellow, 99431");
//  . . .
// automobiles leaving the driveway
driveWay.poll();
driveWay.poll();
driveWay.poll();
driveWay.poll();
```

5.

```
Map<Double, String> studentGPA = new HashMap<>();
studentGPA.put(3.4, "Suzie");
studentGPA.put(3.1, "Ayylah");
studentGPA.put(2.9, "Billi");
studentGPA.put(3.4, "Anna");
studentGPA.put(3.1, "Arlough");
```

6.

```
String name = "bookAuthor";
HashSet<Character> letterSet = new HashSet<>();

// Store name in collection
```

```
for (Character c: name.toCharArray())
{
    letterSet.add(c);
}

// Print letters out to print the name
// "bookAuthor"
for (Character c: letterSet)
{
    System.out.print(c + " ");
}
```

7.

```
// Stack duplication

Stack<Integer> firstStack = new Stack<>();
Stack<Integer> secondStack = new Stack<>();
for (int i = 0; i < 10; i++)
    firstStack.push(i);

// Copy firstStack into secondStack
for (int i = 0; i< firstStack.size(); i++)
    secondStack.push(firstStack.pop());
```

8.

```
Queue<String> nameQueue = new ArrayDeque<>();
nameQueue.offer("Billi");
nameQueue.offer("Sarah");
nameQueue.offer("Caylee");
nameQueue.offer("Linwood");

Iterator<Queue<String>> qIter = new Iterator(Queue.Iterator);
while (qIter.next())
    System.out.println("Name: " + qIter.hasNext());
```

9.

```
Set<String> studentGPA = new HashSet<>();
studentGPA.put(3.4, "Suzie");
studentGPA.put(3.1, "Ayylah");
studentGPA.put(2.9, "Billi");
studentGPA.put(3.4, "Anna");
studentGPA.put(3.1, "Arlough");
```

10.

```
for (int i=0; i<10; i++)
{
    Map<String, Integer> numList = new Map<>();
    numList.put(" ", i);
}
```

Think about It

1. Why do you think the developers of the Java language decided to group the collections classes examined in this chapter together under a single framework?
2. In plain language and in the context of programming, what is a collection? What is a data structure?
3. How does the object-oriented concept of an interface help improve the usability of the Java `Collections` framework?
4. What is the general overall structure of the Java `Collections` framework? What interfaces are present? What classes?

5. When is it to the advantage of the developer to use an ArrayList<E> object? When might a standard Java array be more useful than an ArrayList<E>?
6. What are some of the static methods of the Collections class? Do they work on objects instantiated from all classes in the Java Collections framework?
7. What are some static methods of the Collections class that if you, the developer, tried to implement them manually would take quite an effort?
8. Which classes in the Java Collections framework are capable of index notation? Which are not, and for those, how do they "map" the elements in their collections?
9. Which classes in the Java Collections framework allow duplicate elements to be inserted? Which do not? For those that do not, why do they prevent this from happening?
10. What are several benefits of using a LinkedList<E> data structure? What interface in the Java Collections framework does it implement?
11. How does a Stack<E> manage its elements differently than, say, the ArrayList<E> or LinkedList<E> classes?
12. What is meant by the data-handling acronym "LIFO"? What is meant by "FIFO"?
13. What are some real-world uses of a Stack<E>?
14. How does a Queue<E> data structure manage its elements differently than a Stack<E>?
15. What are some real-world uses of a Queue<E>?
16. What benefits mentioned by the official Java documentation are gained by using an ArrayDeque<E> object in situations where both a stack and a queue are needed?
17. How is a PriorityQueue<E> similar to a "regular" queue data structure? How does it behave differently? How can this different behavior be useful in real-world applications?
18. How does the data handling performed by a Set<E> prevent it from accepting duplicate values?
19. What is a major characteristic/benefit of the HashSet<E>?
20. What is a major characteristic/benefit of the TreeSet<E>?
21. What interface must a class implement in order for its objects to work naturally with the behavior of a TreeSet<E> collection?
22. What are the benefits of using an Iterator<E> object with the classes of the Java Collections framework?
23. How is an Iterator<E> object created for any particular collection class object?
24. How do the classes that implement the Map<K,V> interface differ from other classes in the Java Collections framework?
25. How does a LinkedHashMap<K,V>'s behavior differ from that of the HashMap<K,V>?
26. How does TreeMap<K,V> differ from HashMap<K,V>?
27. If you needed to store multiple values with a particular key in a Map<K,V> structure, how could you accomplish this?

Short Syntax Problems

1. Write a small program that will prompt the user to enter a word. The program will capture the word and store its characters in a Stack<E> object. Write the logic that will check to see if the word entered is a palindrome (i.e., a word that is spelled the same forward and backward—e.g., "racecar," "madam," "level," etc.). *Hint:* Create as many other Stack<E> objects as you might need to accomplish this.
2. Write a small program that performs the same task and logic as in short syntax problem #1 but uses a LinkedList<E> to accomplish the task.
3. Write a small program that will prompt to the console for a large block of text. (You can copy/paste a large paragraph from the web, etc. Be careful not to include special characters!) The program will capture the large block and then proceed to count the occurrences of words in the block. Store the counts in a Map<K,V> object.
4. Write a small program that will prompt the user to enter the names of family members and their ages. Allow the program the loop until an age of -1 is entered, which will end the loop. Store the ages and family member names in a TreeMap<K,V>. Print the names and ages out along with a category label. For example, if the family member is eight years old, print "< 10 Years Old" and print additional family members until an age of less than twenty is encountered, then print "< 20 Years Old," and so on for each grouping of family members' ages.
5. Write a **method** that will accept as an input parameter a Stack<E> and return back a reference to a new Stack<E> object that contains the elements from the parameter stack but in reverse.

Full Problems

1. Write a program that is similar to short syntax problem #3, but alter it in the following ways:
   a. The program will count the occurrence of letters and characters (punctuation, etc.) in the block of text.
   b. Store the counts in a `HashMap<K,V>`.
   c. When the block of text has been processed, print to the console a visual representation of a histogram. Print the character counted, and print a number of asterisks equal to the count to the right of the letter. For example,

```
e: ***************
a: ************
o: ******
. . .
```

   d. Print the histogram in order of the most frequently occurring letter first, second next, and so on.
2. Create a student management system for a small university that will use an appropriate `Map<K,V>` data structure as its core data-storing object along with any other needed `Collections` framework data structures. The application will function as follows:
   a. A menu system will be presented to the user giving them the option to "Create a Course" and "Add Students to a Course."
   b. When creating a course, the user will be asked for a code for the course and the course name. For example, "`19929`" might be a code, and "`Introduction to Systems Analysis`" might be the name.
   c. When adding a student to a course, the application should allow the user to choose a course to add students to. Once a course is chosen, the user can type in first/last names for the students to add to the course. The user will type the word "done" when finished adding students.
   d. The same student can be added to multiple courses as the user populates the courses.
   e. The user can choose a menu option to print a report showing by course and all students in each course.
   f. The user can choose a menu option to print a report showing by students and all the courses each student is currently "enrolled" in.