Java Object-Oriented Concepts Unit

Lesson 3: Collections and Maps

SOFTWARE GUILD

The Learning House

427 S 4th Street #300

Louisville KY 40202

Lesson 3: Collections and Maps

Overview

This lesson is a quick introduction to the Java Collections Framework and the Map interface. Collections and Maps are data structures that allow you to store values and objects in a more flexible manner than arrays. We will concentrate on Lists and Maps.

Java Collections Framework

In the broadest sense, a collection is simply a data structure that groups multiple elements together into a single unit. We make extensive use of collections in Java to manipulate groups of related data. Collections usually represent a group of related data such as a class of students, a group of addresses, or a list of classes in a school schedule.

The Java Collections Framework is a library of code that allows programmers to manipulate groups of related objects. One benefit of using the Collections Framework is that it prevents us from having to "reinvent the wheel" for collections manipulation because the Framework is available to all Java programmers. Because it is widely used, the implementation of these algorithms has been honed over time to be fast, efficient, and bug-free. Another great benefit of the Framework is that there is one Application Programming Interface (API) that is widely used by Java programmers. This means that you can easily read and understand someone else's Collections Framework code when working on new projects or when helping someone else with an issue.

Javadoc

The best place to learn about the structure and capabilities of the Collection Framework and Map interface is the Javadoc. Javadoc is HTML-formatted code documentation that is generated from special Javadoc tags in the comments of code (we'll learn about how to generate our own Javadocs later in the course). All of the code that comes with the JDK has extensive Javadocs available online — this should be the first place you look to see how a particular method or interface is supposed to work.

Collections Framework Structure

In-depth examination of the Collections Framework Javadoc is left as an exercise for the reader (we will also go a little deeper in class), but there are a few things that you should know about the Collections Framework:

- The framework is made up of several interfaces and several implementing classes (we'll learn more about interfaces in coming lessons).
- The underlying implementations have different characteristics. For example, Sets cannot have duplicate entries, but Lists can.
- All implementations of the Collection interface share the following methods:
    a. add
    b. addAll
    c. clear
    d. contains
    e. containsAll

f. equals
g. hashCode
h. isEmpty
i. iterator
j. remove
k. removeAll
l. retainAll
m. size
n. toArray

- Some implementations of the Collection interface have additional methods. For example, Lists have these additional methods:
    a. get
    b. indexOf
    c. lastIndexOf
    d. listIterator
    e. set
    f. subList

Explore the Javadoc in detail to gain a full understanding of the Collections Framework.

Iterator

An iterator is an object that allows you to visit each element in a collection individually. All iterators have a **hasNext** method that returns true if there are more elements to visit and a **next** method that retrieves the next element. Some iterators also implement the **remove** method which is used to remove elements from the collection during iteration, but this method is not required. We will see examples of using an iterator to visit all the elements in a collection in the following sections.

Generics

The definition of a generic type from Oracle is:

> A generic class or interface that is parameterized over types.

So, what does that mean? One way to illustrate this is with Collections (below we'll see specific examples with ArrayLists and HashMaps). As mentioned above, a collection is simply a container that holds related objects. Generic types allow us to specify what kind of objects we can place in our container. For example, we can specify that our container only accepts Strings — if we tried to add anything other than Strings to our container, an error will occur. In earlier versions of Java, Collections were not generic types and you could put any type of object into a Collection. This meant that you had to check each time you retrieved a value from your collection to ensure that it was the type you were expecting. You can still create collection like this, but you will find that it is almost always advantageous to create containers that only accept a particular type — it

makes your code cleaner, smaller, and easier to maintain. The List and Map examples below all use generic types.

List

A List is an ordered collection of items. As mentioned above, Lists may include duplicate elements. Some of the operations that stand out in the List interface are:

1. Access to elements based on position in the list
2. The ability to search for elements in the list (indexOf and lastIndexOf)
3. The ability to retrieve a sublist

ArrayList

The particular implementation of List that we will concentrate on for now is the ArrayList. Like any implementation of the List interface, the ArrayList has all of the features of the Collection interface and all of the features of the List interface. One way to think about the ArrayList is like an array on steroids. The main convenience that ArrayLists have over arrays is that you can dynamically add and remove elements from an ArrayList and it will automatically resize for you. Below are some code snippets that illustrate some of the features of the ArrayList:

Creation and Adding Elements:

```
public static void main(String[] args) {
    // create an ArrayList of String objects
    ArrayList<String> stringList = new ArrayList<>();

    // ask the list how big it is
    System.out.println("List size before adding any Strings: " + stringList.size());

    // add a String object to our list
    stringList.add("My First String");

    // ask the list how big it is
    System.out.println("List size after adding one String: " + stringList.size());

    // add another String object to our list
    stringList.add("My Second String");

    // ask the list how big it is
    System.out.println("List size after adding two Strings: " + stringList.size());
}
```

Removing Elements

```
public static void main(String[] args) {
    // create an ArrayList of String objects
    ArrayList<String> stringList = new ArrayList<>();

    // add a String object to our list
    stringList.add("My First String");

    // add another String object to our list
    stringList.add("My Second String");

    // ask the list how big it is
    System.out.println("List size after adding two Strings: " + stringList.size());

    // remove the second String object from our list - remember that our
    // indexes start counting at 0 instead of 1
    stringList.remove(1);

    // ask the list how big it is
    System.out.println("List size after removing one String: " + stringList.size());

    // remove the remaining String object from our list - remember that
    // the list resizes automatically so if there is only one element
    // in a list it is alway at index 0
    stringList.remove(0);

    // ask the list how big it is
    System.out.println("List size after removing last String: " + stringList.size());

    // what happens if you try to remove another element??? Give it a
    // try...
}
```

Visiting All Elements: Enhanced For Loop

```
public static void main(String[] args) {
    // create an ArrayList of String objects
    ArrayList<String> stringList = new ArrayList<>();

    // add a String object to our list
    stringList.add("My First String");

    // add another String object to our list
    stringList.add("My Second String");

    // add another String object to our list
    stringList.add("My Third String");

    // add another String object to our list
    stringList.add("My Fourth String");

    // ask the list how big it is
    System.out.println("List size: " + stringList.size());

    // print every String in our list with an enhanced for loop
    for (String s : stringList) {
        System.out.println(s);
    }
}
```

Visiting All Elements: Iterator

```
public static void main(String[] args) {
    // create an ArrayList of String objects
    ArrayList<String> stringList = new ArrayList<>();

    // add a String object to our list
    stringList.add("My First String");

    // add another String object to our list
    stringList.add("My Second String");

    // add another String object to our list
    stringList.add("My Third String");

    // add another String object to our list
    stringList.add("My Fourth String");

    // ask the list how big it is
    System.out.println("List size: " + stringList.size());

    // print every String in our list with an iterator

    // ask for the iterator - we must ask for an iterator of Strings
    // What happens if we don't???
    Iterator<String> iter = stringList.iterator();

    // get String objects from the list while there are still Strings
    // remaining
    while(iter.hasNext()) {
        String current = iter.next();
        System.out.println(current);
    }
}
```

Map Interface

A Map is an object that maps keys to values. A Map models the concept of a mathematical function. In a Map, each key can map to one, and only one, value; so it cannot contain duplicate keys. The Map interface is not a member of the Collections Framework; however, we can get a collection view of either the keys or the values of a Map. One thing you must keep in mind when you get a collection of either the keys or the values is that it will be returned as a Set, and the order in which items are returned from a Set is not guaranteed. Although Lists and Sets are both Collections, Lists are inherently ordered and Sets are not.

HashMap

We will focus on the HashMap implementation of the Map interface for now. HashMap is the most commonly used implementation of Map interface. Below are some code snippets illustrating some of the features of the HashMap:

Creation and Adding Key/Value Mapping

```
public static void main(String[] args) {
    // create a map that maps a country to its population
    HashMap<String, Integer> populations = new HashMap<>();

    // add the first country
    populations.put("USA", 313000000);

    // add the next country
    populations.put("Canada", 34000000);

    // add another country
    populations.put("United Kingdom", 63000000);

    // add another country
    populations.put("Japan", 127000000);

    // ask the map for its size
    System.out.println("Map size is: " + populations.size());
 }
```

Replacing a Key/Value Mapping

```
public static void main(String[] args) {
    // create a map that maps a country to its population
    HashMap<String, Integer> populations = new HashMap<>();

    // add the first country
    populations.put("USA", 200000000);

    // add the next country
    populations.put("Canada", 34000000);

   // add another country
   populations.put("United Kingdom", 63000000);

   // add another country
    populations.put("Japan", 127000000);

    // replace the mapping for population of the USA - original
    // number was too low
    populations.put("USA", 313000000);

    // ask the map for its size - it will be 4
    System.out.println("Map size is: " + populations.size());
}
```

Retrieving a Key/Value Mapping

```
public static void main(String[] args) {
    // create a map that maps a country to its population
    HashMap<String, Integer> populations = new HashMap<>();

    // add the first country
    populations.put("USA", 313000000);

    // add the next country
    populations.put("Canada", 34000000);

    // add another country
    populations.put("United Kingdom", 63000000);

    // add another country
    populations.put("Japan", 127000000);

    // ask the map for its size - it will be 4
    System.out.println("Map size is: " + populations.size());

    // get the poplation of Japan and print it to the screen
    Integer japanPopulation = populations.get("Japan");
    System.out.println("The population of Japan is: " + japanPopulation);
}
```

Listing all the Keys

```
public static void main(String[] args) {
    // create a map that maps a country to its population
    HashMap<String, Integer> populations = new HashMap<>();

    // add the first country
    populations.put("USA", 313000000);

    // add the next country
    populations.put("Canada", 34000000);

    // add another country
    populations.put("United Kingdom", 63000000);

    // add another country
    populations.put("Japan", 127000000);

    // ask the map for its size - it will be 4
    System.out.println("Map size is: " + populations.size());

    // get the Set of keys from the map
    Set<String> keys = populations.keySet();

    // print the keys to the screen - is the order they are printed
    // what you would expect?
    for (String k : keys) {
        System.out.println(k);
    }
}
```

Listing all the Values Key by Key

```
public static void main(String[] args) {
    // create a map that maps a country to its population
    HashMap<String, Integer> populations = new HashMap<>();

    // add the first country
    populations.put("USA", 313000000);

    // add the next country
    populations.put("Canada", 34000000);

    // add another country
    populations.put("United Kingdom", 63000000);

    // add another country
    populations.put("Japan", 127000000);

    // ask the map for its size - it will be 4
    System.out.println("Map size is: " + populations.size());

    // get the Set of keys from the map
    Set<String> keys = populations.keySet();

    // print the keys and associated values to the screen
    for (String k : keys) {
        System.out.println("The population of " + k + " is " + populations.get(k));
    }
}
```

Listing all the Values: Value Collection

```
public static void main(String[] args) {
    // create a map that maps a country to its population
    HashMap<String, Integer> populations = new HashMap<>();

    // add the first country
    populations.put("USA", 313000000);

    // add the next country
    populations.put("Canada", 34000000);

    // add another country
    populations.put("United Kingdom", 63000000);

    // add another country
    populations.put("Japan", 127000000);

    // ask the map for its size - it will be 4
    System.out.println("Map size is: " + populations.size());

    // get the Collection of values from the Map
    Collection<Integer> popValues = populations.values();

    // list all of the population values - how can we tell which population
    // value goes with each country?
    for (Integer p : popValues) {
        System.out.println(p);
    }
}
```