Week 08 - Collections

Louis Botha, `louis.botha@tuni.fi`

Java Collections Class | Common Methods of Java Collections Class

Java Collections are a group of classes and interfaces that provide a framework for storing and manipulating groups of objects in Java. The framework provides a set of reusable data structures such as lists, sets, maps, and queues to manage collections of objects in Java.

Some of the commonly used Java collections classes and interfaces include:

1. **List Interface**: This interface extends the Collection interface and is used to store an ordered collection of elements. The most commonly used classes implementing this interface are ArrayList and LinkedList.

2. **Set Interface**: This interface extends the Collection interface and is used to store a collection of unique elements. The most commonly used classes implementing this interface are HashSet and TreeSet.

3. **Map Interface**: This interface is used to store a collection of key-value pairs, where each key is associated with a unique value. The most commonly used classes implementing this interface are HashMap and TreeMap.

4. **Queue Interface**: This interface is used to store a collection of elements in which the element entered first is the first to be removed (FIFO). The most commonly used classes implementing this interface are LinkedList and PriorityQueue.

5. **Stack Class**: This class represents a last-in, first-out (LIFO) stack of objects.

Java collections provide a number of useful methods to manipulate, iterate and sort the elements in the collections. The collections framework is also highly extensible, allowing developers to create custom collections that suit their specific needs.

Java collections are an important part of the Java language and are widely used in the development of Java applications. Understanding the collections framework and its various classes and interfaces is crucial for any Java developer.

We wil

The List interface

The `List` interface in Java is an ordered collection of elements that can contain duplicate elements. It extends the Collection interface and provides methods to access elements by their index or position in the list.

Some of the important methods provided by the List interface are:
1. `add(E element)`: Adds the specified element to the end of the list.
2. `add(int index, E element)`: Inserts the specified element at the specified position in the list.
3. `remove(int index)`: Removes the element at the specified position in the list.
4. `get(int index)`: Returns the element at the specified position in the list.
5. `set(int index, E element)`: Replaces the element at the specified position in the list with the specified element.
6. `size()`: Returns the number of elements in the list.

7. `indexOf(Object o)`: Returns the index of the first occurrence of the specified element in the list.

The most commonly used classes implementing the List interface are ArrayList and LinkedList. The ArrayList class is backed by an array and provides fast random access to elements, while the LinkedList class provides efficient insertion and deletion of elements, especially for large lists.

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

List interface is often used in situations where elements need to be maintained in a specific order and duplicates are allowed. It provides many useful methods for manipulating the elements in the list, such as sorting, searching, and modifying.

In summary, the List interface in Java provides a powerful tool for managing ordered collections of elements, and is an essential component of the Java collections framework.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
   public static void main(String[] args) {
      // Create an ArrayList to store a list of integers
      List<Integer> myList = new ArrayList<Integer>();

      // Add some integers to the list
      myList.add(10);
      myList.add(20);
      myList.add(30);
```

```
        myList.add(40);

        // Print the list to the console
        System.out.println("My list of integers: " + myList);

        // Add an integer to the beginning of the list
        myList.add(0, 5);

        // Remove an integer from the list
        myList.remove(2);

        // Print the list to the console again
        System.out.println("My updated list of integers: " + myL
ist);

        // Get the value at a specific index
        int value = myList.get(1);
        System.out.println("The value at index 1 is: " + value);

        // Check if the list contains a specific value
        boolean containsValue = myList.contains(30);
        System.out.println("Does the list contain the value 30?
" + containsValue);

        // Get the size of the list
        int size = myList.size();
        System.out.println("The size of the list is: " + size);

        // Print the list
        System.out.print("The list: ");
```

```
      myList.forEach((num) -> System.out.print(num + " "));
}
}

// Output
My list of integers: [10, 20, 30, 40]
My updated list of integers: [5, 10, 30, 40]
The value at index 1 is: 10
Does the list contain the value 30? true
The size of the list is: 4
The list: 5 10 30 40
```

Here is an example using an object created from a custom class.

```
import java.util.ArrayList;
import java.util.List;

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
```

```
        return age;
    }
}

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

        // Create an ArrayList to store a list of Person objec
ts
        List<Person> people = new ArrayList<Person>();

        // Add some Person objects to the list
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));
        people.add(new Person("David", 40));

        // Print the list to the console
        System.out.println("My list of people");
        people.forEach((person)
            -> System.out.println(person.getName() + " - " + p
erson.getAge()));

        // Add a new Person object to the list
        people.add(new Person("Emily", 20));

        // Remove a Person object from the list
        people.remove(2);

        // Get the name of a specific Person object in the lis
t
```

```
        String name = people.get(1).getName();
        System.out.println("The name of the person at index 1 is: " + name);

        // Check if the list contains a specific Person object
        boolean containsPerson = people.contains(new Person("Bob", 25));
        System.out.println("Does the list contain a person with name 'Bob' and age 25? " + containsPerson);

        // Get the size of the list
        int size = people.size();
        System.out.println("The size of the list is: " + size);

        // Print the list to the console again
        System.out.println("My updated list of people: ");
        people.forEach((person)
            -> System.out.println(person.getName() + " - " + person.getAge()));
    }
}

// Output
My list of people
Alice - 30
Bob - 25
Charlie - 35
David - 40
The name of the person at index 1 is: Bob
```

```
Does the list contain a person with name 'Bob' and age 25? fal
se
The size of the list is: 4
My updated list of people:
Alice - 30
Bob - 25
David - 40
Emily - 20
```

The Set Interface

The `Set` interface represents a collection that contains **no duplicate** elements. In other words, each element in a `Set` must be unique. `Set` is a subtype of the `Collection` interface and inherits all of its methods.

Here are some important characteristics of the `Set` interface:

- Sets cannot contain duplicate elements. If you attempt to add a duplicate element to a set, the second addition will be ignored and the set will remain unchanged.
- The order of elements in a set is not guaranteed. Implementations of the `Set` interface may choose to order elements in a specific way, but the API does not specify any particular ordering.
- The `Set` interface does not provide methods for accessing elements by index, since the order of elements is not guaranteed. However, you can use an iterator to iterate over the elements of a set.

Some of the most commonly used methods of the `Set` interface include:

- `add(E e)`: Adds the specified element to the set if it is not already present.
- `remove(Object o)`: Removes the specified element from the set if it is present.
- `contains(Object o)`: Returns `true` if the set contains the specified element, and `false` otherwise.
- `size()`: Returns the number of elements in the set.
- `isEmpty()`: Returns `true` if the set contains no elements, and `false` otherwise.

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

Here's an example of how you might use a `Set` in Java:

```
import java.util.HashSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {
        // Create an HashSet to store a list of names(Strings)
        Set<String> names = new HashSet<>();

        // Add to the elements to the set
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // This addition will be ignored, since "Alice" is alr
eady in the set
        names.add("Alice");

        // prints "Size of set: 3"
        System.out.println("Size of set: " + names.size());

        // prints "Is Bob in the set? true"
        System.out.println("Is Bob in the set? " + names.conta
ins("Bob"));
```

```
        // Removes the name(String) from the set
        names.remove("Charlie");

        // prints "Updated size of set: 2"
        System.out.println("Updated size of set: " + names.siz
e());

        // Print the set to the console
        System.out.println("My updated set of people: ");
        for (String name : names) {
            System.out.println(name);
        }
    }
}

//Output
Size of set: 3
Is Bob in the set? true
Updated size of set: 2
My updated set of people:
Bob
Alice
```

More advanced example using a custom object

```
import java.util.HashSet;
import java.util.Set;

class Person {
    private String name;
    private int age;
```

```
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Implement the equals method for comparing the person ob
ject
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Person person = (Person) o;
        return this.name.equals(person.getName());
    }

    // Implement the hashCode method for generating a hash of
the object based on the name property
```

```
    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
public class SetObjectExample {
    public static void main(String[] args) {
        Set<Person> personSet = new HashSet<>();

        // Add some people to the set
        Person alice = new Person("Alice", 30);
        personSet.add(alice);
        Person bob = new Person("Bob", 25);
        personSet.add(bob);
        Person charlie = new Person("Charlie", 40);
        personSet.add(charlie);

        // Add a duplicate person
        Person alice2 = new Person("Alice", 30);
        personSet.add(alice2);

        // The size of the set should be 3, since Alice is a d
uplicate
        System.out.println("Size of set: " + personSet.size
());

        // Check if Bob is in the set
        if (personSet.contains(bob)) {
            System.out.println("Bob is in the set");
        }
```

```
        // Remove Charlie from the set
        personSet.remove(charlie);

        // Print out the remaining people in the set
        System.out.println("People in set:");
        for (Person person : personSet) {
            System.out.println(person.getName() + ", age " + p
erson.getAge());
        }

        // Clear the set
        personSet.clear();
        System.out.println("Size of set after clearing: " + pe
rsonSet.size());
    }
}

//Output
Size of set: 3
Bob is in the set
People in set:
Bob, age 25
Alice, age 30
Size of set after clearing: 0
```

The `hashCode()` method is used to generate a hash code for an object, which is an integer that is used to identify the object in hash-based collections like `HashSet` and `HashMap`.

When you add an object to a `HashSet`, the `HashSet` implementation calls the `hashCode()` method of the object to determine which bucket to put the object in. If two objects have the same hash code, they are considered equal and only one of them will be added to the set.

In the case of the `hashCode()` method, it is important to override it in order to ensure that objects that are considered equal have the same hash code. If you don't override `hashCode()`, then the default implementation from the `Object` class will be used, which returns a different hash code for every object.

By overriding `hashCode()` to use the `name` field of the `Person` class, we ensure that two `Person` objects with the same name will have the same hash code, and will be considered equal by the `HashSet`.

The Map interface

The `Map` interface in Java is a collection that stores key-value pairs, where each key is associated with a unique value. The Map interface does not extend the Collection interface, but provides methods to access, add, remove and manipulate key-value pairs.

Some of the important methods provided by the Map interface are:

1. `put(K key, V value)`: Associates the specified value with the specified key in the map.
2. `get(Object key)`: Returns the value associated with the specified key in the map.
3. `remove(Object key)`: Removes the key-value pair associated with the specified key from the map.
4. `containsKey(Object key)`: Returns true if the map contains the specified key.
5. `containsValue(Object value)`: Returns true if the map contains the specified value.
6. `keySet()`: Returns a Set view of the keys in the map.
7. `values()`: Returns a Collection view of the values in the map.
8. `entrySet()`: Returns a Set view of the key-value pairs in the map.

The most commonly used classes implementing the Map interface are HashMap and TreeMap. HashMap is based on a hash table and provides constant-time performance for most operations, while TreeMap is based on a tree structure and maintains the elements in sorted order.

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

HashMaps are commonly used when we need to store and retrieve data in key-value pairs, and we need to do it with constant time complexity. It provides a fast and efficient way to access and manipulate key-value pairs.

Here are some examples of good use cases for HashMaps:

1. Caching - In applications that need to frequently access data from a database or an external API, it's often faster to store the data in a HashMap and access it from memory rather than making repeated requests to the external source. HashMaps can be used as a cache to store the data and quickly retrieve it when needed.
2. Indexing - HashMaps can be used to build indexes for large datasets, making it faster to search and retrieve data based on certain criteria.
3. Counting - HashMaps can be used to count the occurrences of elements in a collection. For example, you could use a HashMap to count the frequency of words in a document.
4. Configuration - HashMaps can be used to store configuration settings for an application, where the keys represent the configuration names and the values represent the corresponding values.
5. Memoization - In dynamic programming, HashMaps can be used to store intermediate results of computations to avoid redundant calculations, leading to faster execution times.

In summary, the Map interface in Java provides a powerful tool for managing collections of key-value pairs, and is an essential component of the Java collections framework.

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        // create a new HashMap instance with String keys and
Integer values
        HashMap<String, Integer> map = new HashMap<>();

        // add some key-value pairs to the map
        map.put("Alice", 25);
        map.put("Bob", 32);
        map.put("Charlie", 28);

        // retrieve the value for a given key
        int aliceAge = map.get("Alice");
        System.out.println("Alice's age is " + aliceAge);

        // check if a key is present in the map
        boolean hasBob = map.containsKey("Bob");
        System.out.println("The map " + (hasBob ? "has" : "doe
s not have") + " a value for Bob");

        // iterate over the key-value pairs in the map
        for (String key : map.keySet()) {
            int value = map.get(key);
            System.out.println(key + " is " + value + " years
old");
        }
    }
}
```

In this example, we create a new `HashMap` instance with `String` keys and `Integer` values. We add three key-value pairs to the map, with keys `"Alice"`, `"Bob"`, and `"Charlie"`, and values `25`, `32`, and `28`, respectively.

We then retrieve the value for the key `"Alice"` using the `get()` method and store it in a variable called `aliceAge`. We print out a message to the console showing Alice's age.
Next, we use the `containsKey()` method to check if the key `"Bob"` is present in the map. We print out a message to the console indicating whether or not the map has a value for Bob.

Finally, we use a `for` loop to iterate over the key-value pairs in the map. For each key in the map, we retrieve the corresponding value using the `get()` method and print out a message to the console showing the key and value. This allows us to print out a list of all the people in the map and their ages.

Overall, this example demonstrates some of the basic functionality of the `HashMap` class in Java, including adding and retrieving key-value pairs, checking for the presence of keys, and iterating over the key-value pairs in the map.

```
import java.util.HashMap;
import java.util.Map;

public class AdvancedHashMapExample {
    public static void main(String[] args) {
        // create a new HashMap instance with String keys and
Double values
        HashMap<String, Double> salaries = new HashMap<>();

        // add some key-value pairs to the map
        salaries.put("Alice", 50000.0);
        salaries.put("Bob", 75000.0);
        salaries.put("Charlie", 60000.0);
```

```
        // print out the salaries of all employees
        System.out.println("Initial salaries:");
        for (String name : salaries.keySet()) {
            double salary = salaries.get(name);
            System.out.println(name + ": " + salary);
        }

        // increase the salaries of all employees by 10%
        for (String name : salaries.keySet()) {
            double salary = salaries.get(name);
            salaries.put(name, salary * 1.1);
        }

        // print out the new salaries of all employees
        System.out.println("New salaries:");
        for (Map.Entry<String, Double> entry : salaries.entryS
et()) {
            String name = entry.getKey();
            double salary = entry.getValue();
            System.out.println(name + ": " + salary);
        }

        // remove the lowest-paid employee from the map
        String lowestPaid = null;
        double lowestSalary = Double.MAX_VALUE;
        for (Map.Entry<String, Double> entry : salaries.entryS
et()) {
            String name = entry.getKey();
            double salary = entry.getValue();
            if (salary < lowestSalary) {
```

```
                lowestPaid = name;
                lowestSalary = salary;
            }
        }
        salaries.remove(lowestPaid);

        // print out the updated salaries of all employees
        System.out.println("Updated salaries:");
        for (Map.Entry<String, Double> entry : salaries.entryS
et()) {
            String name = entry.getKey();
            double salary = entry.getValue();
            System.out.println(name + ": " + salary);
        }
    }
}
```

In this example, we create a `HashMap` with `String` keys and `Double` values, representing the salaries of different employees. We add three key-value pairs to the map, representing the salaries of Alice, Bob, and Charlie.

We then use a `for` loop to iterate over the keys in the map and print out the initial salaries of all employees.

Next, we use another `for` loop to increase the salaries of all employees by 10%. We retrieve the value for each key using the `get()` method, multiply it by 1.1, and then store the new value using the `put()` method.

We then use a `for` loop with `Map.Entry` objects to iterate over the key-value pairs in the map and print out the new salaries of all employees.

Finally, we use another `for` loop to find the lowest-paid employee in the map and remove them from the map using the `remove()` method. We then use another `for` loop to print out the updated salaries of all employees.

This example demonstrates some additional functionality of the `HashMap` class, including iterating over key-value pairs using `Map.Entry` objects, performing calculations on the values stored in the map, and removing key-value pairs from the map.

Collections Read More