



Java Programming

6-3

Collections – Part 2



Overview

This lesson covers the following topics:

- Implement a HashMap
- Implement a stack by using a deque
- Define a link list
- Define a queue
- Implement a comparable interface



Collections

- We have seen previously that we can use ArrayLists and HashSets to store multiple data.
- Java has a rich library of collections that will handle a varied list of requirements.



Maps

- A map is a collection that links a key to a value.
- Similar to how an array links an index to a value, a map links a key (one object) to a value (another object).
- Maps, like sets, cannot contain duplicates.
- This means each key can only exist once and can only link to a single value.
- Since Map is an interface, you must use one of the classes that implement Map such as HashMap to instantiate a map.

HashMaps

- HashMaps are maps that link a Key to a Value.
- The Key and Value can be of any type, but their types must be consistent for every element in the HashMap.
- Below is a generic breakdown of how to initialize a HashMap.

```
HashMap<KeyType,ValueType> mapName = new HashMap<KeyType,ValueType>();
```

HashMaps fruitBowl Example

- For example, we wish to group together many different fruits and wish to be able to store and later retrieve their color.
- The first step to do this is to initialize a HashMap.

```
HashMap<String,String> fruitBowl = new HashMap<String,String>();
```

Add Fruits to fruitBowl Example

- To add fruits to our fruitBowl, simply use the put(Key,Value) function of HashMaps.
- Add a few fruits to the fruitBowl.
- Each code segment adds the key (fruit name) and value (color) to the HashMap.

```
fruitBowl.put("Apple", "Red");
```

```
fruitBowl.put("Orange", "Orange");
```

```
fruitBowl.put("Banana", "Yellow");
```


get(Key) Method of HashMap

- The Key of a HashMap can be thought of as the index linked to the element, even though it does not necessarily have to be an integer.
- Getting the value stored is easy once we understand that the key is the index: Use the get(Key) method of HashMap.
- To get the color of the Banana in the fruit bowl, use this method which searches through the HashMap until it finds a Key match to the parameter (“Banana”) and returns the Value for that Key (“Yellow”).

```
String bananaColor = fruitBowl.get("Banana");
```

More HashMap Methods

Method	Method Description
boolean containsKey(Object Key)	Returns true if the HashMap contains the specified Key.
boolean containsValue(Object Value)	Returns true if this map maps one or more keys to the specified value.
Set<K> keySet()	Returns a set of the keys contained in the HashMap.
Collection<V> values()	Returns a collection of the values contained in the HashMap.
V remove(Object Key)	Removes the mapping for the specified key from this map if present.
int size()	Returns the number of key-value mappings in the HashMap.



Queues

- A Queue is a list of elements with a first in first out ordering.
- When you enqueue an element, it adds it to the end of the list.
- When you dequeue an element, it returns the element at the front of the list and removes that element from the list.
- For example, picture a line at the movie theater.
- The first person there is the first person to get their ticket (First In First Out, also known as FIFO).



Stacks

- Stacks are Queues that have reverse ordering to the standard Queue.
- Instead of FIFO ordering (like a queue or line at the theater), the ordering of a stack is last in first out.
- This can be represented by the acronym LIFO.

Stack of Pancakes Example

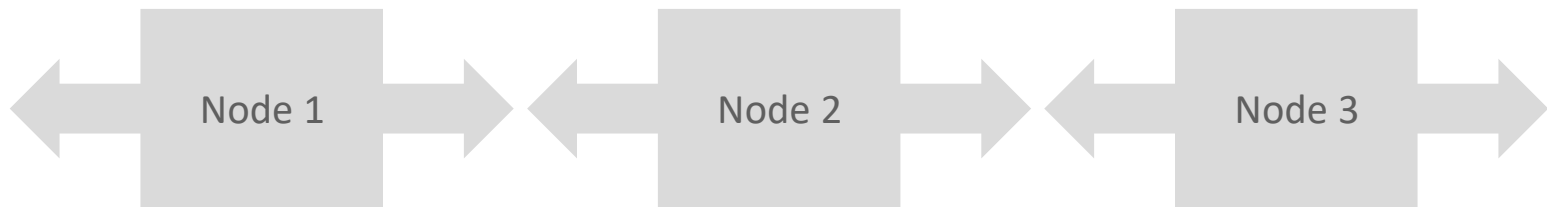
- For example, you have a pile of pancakes.
- Typically this would be called a “stack” of pancakes because the pancakes are added on top of the previous leaving the most recently added pancake at the top of the stack.
- To remove a pancake, you would have to take off the one that was most recently added: The pancake on the top of the stack.
- If you tried to remove the pancake that was added first, you would most likely make a very large mess.

Implementing a Stack: Deque

- One way to implement a Stack is by using a Double-Ended Queue (or deque, pronounced “deck”, for short).
- These allow us to insert and remove elements from either end of the queue using methods inside the Deque class.
- Deques like building blocks, allow you to put pieces on the bottom of your structure or on the top, and likewise pull pieces off from the bottom or top.
- Deques can be implemented by LinkedLists.

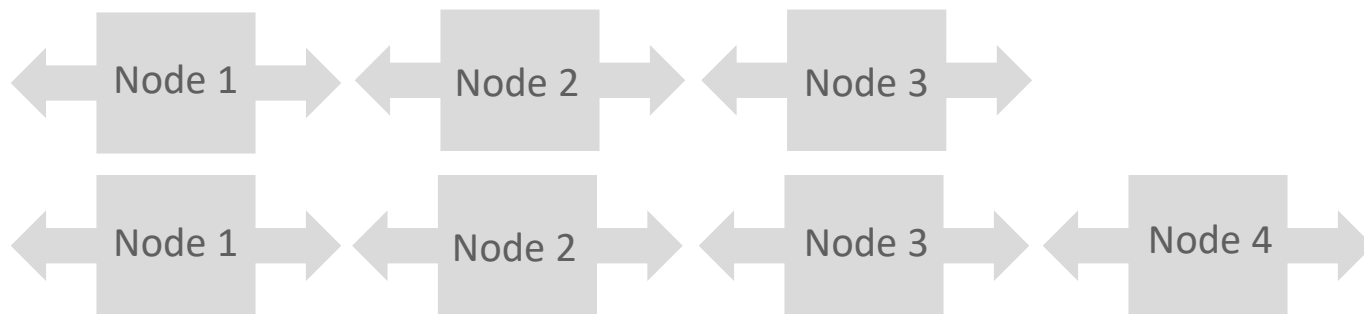
LinkedLists

- A LinkedList is a list of dynamically-stored elements.
- Like an ArrayList, it changes size and has an explicit ordering, but it doesn't use an array to store information.
- It uses an object known as a Node.
- Nodes are like roadmaps: they tell you where you are (the element you are looking at), and where you can go (the previous element and the next element).



Adding Nodes to LinkedLists

- Ultimately, we have a list of Nodes, which point to other Nodes and have an element attached to them.
- To add a Node, set its left Node to the one on its left, and its right Node to the one on its right.
- Do not forget to change the Nodes around it as well.
- A fourth node was added to the end of this linked list:



Initializing a LinkedList

- A LinkedList is initialized in the same way as ArrayList.
- The following code shows how to initialize a LinkedList of pancakes.

```
LinkedList<Pancake> myStack = new LinkedList<Pancake>();
```

LinkedList Methods

FIFO LinkedList Methods	LIFO LinkedList Methods
<code>add(E e)</code> Appends the given element to the end of the list.	<code>add(E e)</code> Appends the given element to the end of the list.
<code>removeFirst()</code> Removes the first element from the list and returns it.	<code>removeLast()</code> Removes the last element from the list and returns it.
<code>getFirst()</code> Returns the first element of the list.	<code>getLast()</code> Returns the last element of the list.

Collection.sort

- We saw earlier to sort a collection with a simple element that we can use `Collections.sort()`.
- In a previous example we saw an `ArrayList` of strings called `groupNames` being sorted by:

```
Collections.sort(groupNames);
```

- This will sort our `ArrayList` in its natural order.

Collection.sort

- This is fine with simple elements but what about our Cell class?
- What if we had additional fields, then which field should it order on?

```
public class Cell {  
    private String data  
    private String data2;  
}
```

Comparable Interface

- For our classes to have a natural order we can implement the interface `java.lang.Comparable`.

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

- So if we implement this interface we must write the code for `compareTo` for our class.



compareTo

- The compareTo method will return an integer based on the following:
- Return a negative value if this object is smaller than the other object
- Return 0 (zero) if this object is equal to the other object.
- Return a positive value if this object is larger than the other object.

Comparable Interface

```
import java.util.Comparator;

public class Cell implements Comparable<Cell> {
    private String data;

    public void set(String celldata)
    {
        data = celldata;
    }
    public String get() {
        return data;
    }
    public int compareTo(Cell c2) {
        if(data.compareTo(c2.get()) < 0 ) return -1;
        if (data.compareTo(c2.get()) == 0) return 0;
        return 1;
    }
}
```

Comparable Interface

- Our compareTo method implementation could have used any or multiple fields from our class.
- We have chosen to simply use the String field called data.

```
public class Cell implements Comparable<Cell> {  
    private String data;  
    <snip>  
    public int compareTo(Cell c2) {  
        if(data.compareTo(c2.get()) < 0 ) return -1;  
        if (data.compareTo(c2.get()) == 0) return 0;  
        return 1;  
    }  
}
```


Cell Driver

- Simple driver class.

- Output

```
Cell c1 = new Cell();  
c1.set("Alison");  
Cell c2 = new Cell();  
c2.set("Brian");
```

```
ArrayList<Cell> list = new ArrayList<Cell>();  
list.add(c2);  
list.add(c1);
```

```
for (Cell c : list) {  
    System.out.println(c.get());  
}
```

```
Collections.sort(list);  
System.out.println("*Sorted*");  
for (Cell c : list) {  
    System.out.println(c.get());  
}
```

```
Brian  
Alison  
*Sorted*  
Alison  
Brian
```

Terminology

Key terms used in this lesson included:

- Deque
- HashMap
- LinkedList
- Node
- Queue
- Stack
- Comparable

Summary

In this lesson, you should have learned how to:

- Implement a HashMap
- Implement a stack by using a deque
- Implement Comparable Interface

