Programming 1

Lecture 11 – Collection Classes

Faculty of Information Technology
Hanoi University

Contents

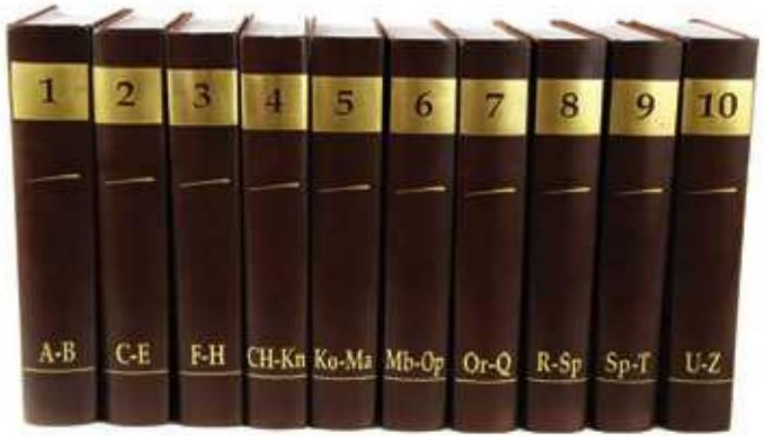
- LinkedList
- HashMap
- HashSet
- TreeSet
- Stack
- Queue

Collection: Overview

- Collections are used to organize multiple objects.
- ArrayList is one of many collection classes which Java provides.
- A collection class provides methods to add, replace or remove elements.
- Some types of collection:
  - List, Set, Map, Stack, Queue

List

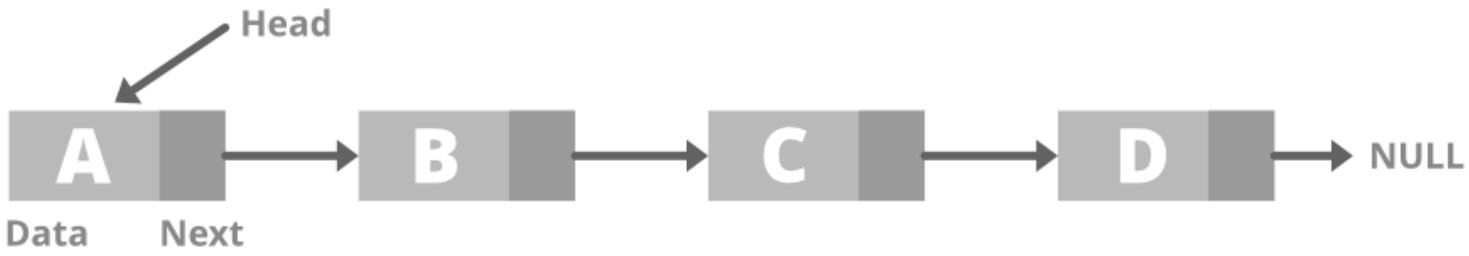
- A List is a collection in which the order of its elements is preserved.
- The same value may occur more than once in a List.



A List of Books

java.util.LinkedList

- A LinkedList (more specifically, singly linked list) is a list in which each element points to the next.
- LinkedList allows efficient addition and removal of elements in the middle of the sequence.
  - It stores its items in **nodes**. It has a link to the first node (the **head**) and each node has a link to the next node.



LinkedList example

```
LinkedList<String> cars = new LinkedList<>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
System.out.println(cars);
```

Program output:

```
[Volvo, BMW, Ford, Mazda]
```

Loop through a LinkedList

- The enhanced for loop can be used to loop through List, Set, Stack, Queue.

```
LinkedList<String> names = new LinkedList<>();
// code omitted
for (String name : names) {
    System.out.println(name);
}
```

Loop through a LinkedList

- Loop through a LinkedList with the regular for loop.
  - The same as looping through an ArrayList.

```
LinkedList<String> cars = new LinkedList<>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
for (int i = 0; i < cars.size(); i++) {
    System.out.println(cars.get(i));
}
```

Notable `LinkedList` methods

- **`addFirst()`**
  - Adds an item to the beginning of the list.
  - Much more efficient at this task than ArrayList.
- **`add(index, element)`**
  - Inserts an element in the middle of the list.
  - Doesn't need to shift elements like ArrayList does.
- **`remove(index)`**
  - Removes an element at a particular position.
  - Doesn't need to shift elements like ArrayList does.

Set

- A set is a collection in which the order of its items is not preserved.
  - Cannot access items by index
  - As a result, a Set is more efficient than a List
- The same object may not occur more than once in a Set.
  - Every item is unique

HashSet example

```
HashSet<String> cars = new HashSet<>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("BMW"); // BMW 2nd time
cars.add("Mazda");
System.out.println(cars);
// no duplicated values in the set
```

Program output:

```
[Volvo, Mazda, Ford, BMW]
```

Items are not listed in the order in which they are added.

Remove an item from HashSet

```
HashSet<String> names = new HashSet<>();
names.add("Romeo");
names.add("Juliet");
// her dad comes and takes her away
names.add("Juliet's Dad");
names.remove("Juliet");
// guess who's with whom now?
System.out.println(names);
```

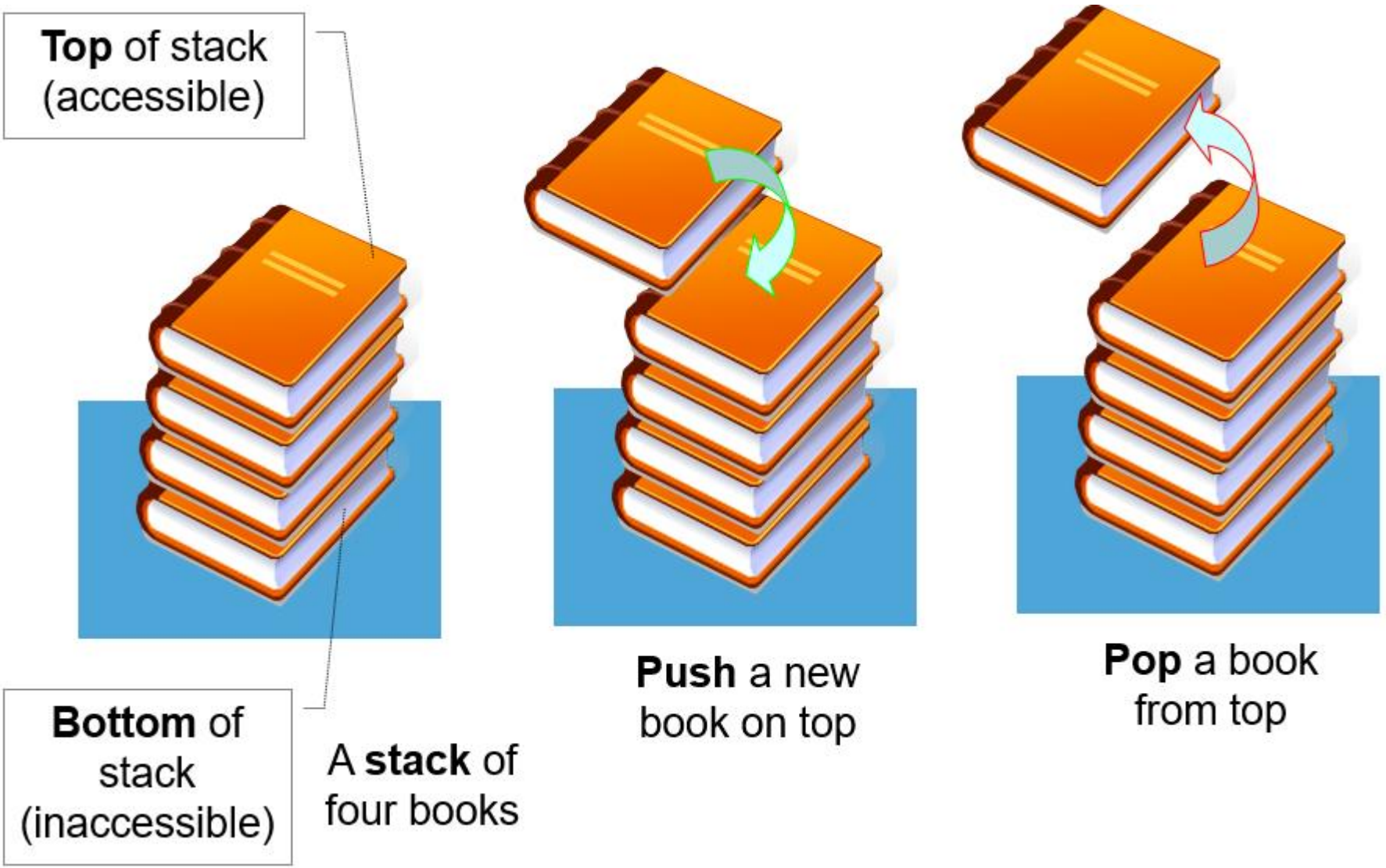
Sorted Set - `TreeSet`

- Elements added to a TreeSet are automatically sorted in ascending order.
  - What if you want to sort in descending order?
- Applications:
  - To build a sorted collection of unique items.
  - To remove duplicates from an existing list.
- Let's see a demo.

Stack

- A Stack is a collection where you can add and remove elements only at the top.
- In a Stack, the order of its elements is preserved.

- A Stack is a **last in, first out** (LIFO) collection.


Top of stack (accessible)
Bottom of stack (inaccessible)
A **stack** of four books
Push a new book on top
Pop a book from top

Stack in Java

- `java.util.Stack`
- Methods:
  - **push**(): adds an element to the top of the stack
  - **pop**(): takes out (returns and removes) the element on top of the stack
  - **peek**(): returns the element on top of the stack without removing it

Stack example

```
Stack<String> s = new Stack<>();
s.push("A");
s.push("B");
s.push("C");
while (s.size() > 0) {
    System.out.print(s.pop() + " ");
}
// Output: C B A
// Stack becomes empty after the loop
```

Queue

- A Queue is a collection where you add items to one end and remove them from the other end.
- A Queue is a **first in, first out** (FIFO) collection.
  - Also called “first come, first served”.

Queue in Java

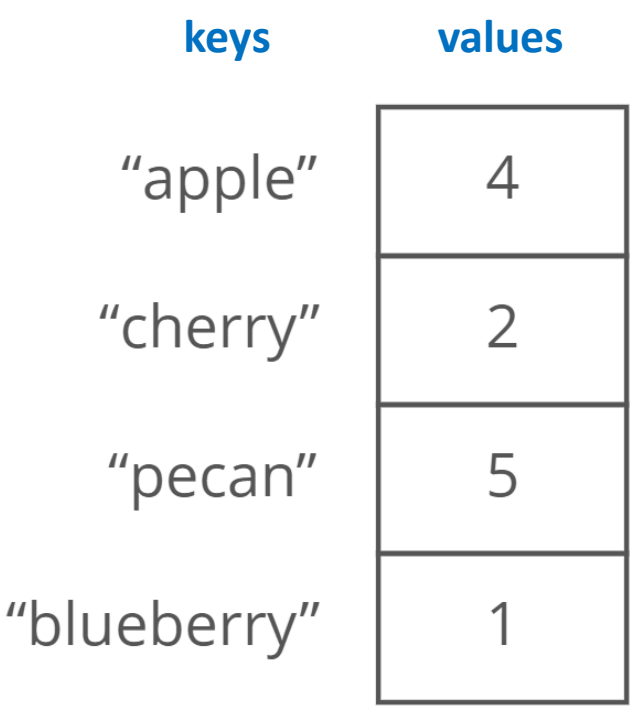
- The LinkedList class implements the Queue interface.
  - We use LinkedList as Queue.
- Methods:
  - **add**(): adds an element to the *tail* of the queue.
  - **remove**(): returns and also removes the element at the *front* of the queue.
  - **peek**(): returns without removing the element at the *front* of the queue.

Queue example

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) {
    System.out.print(q.remove() + " ");
}
// Output: A B C
// Queue becomes empty after the loop
```

Map

- A Map is a collection of **(key, value) pairs** such that each key appears at most once in the collection.
  - Also called **associative array, dictionary** or **lookup table**.
- Unlike arrays (the type of key is restricted to integer), the keys in Map can be any data type.
  - E.g. String, Object…
- In Map, you access a value using a key.



java.util.HashMap

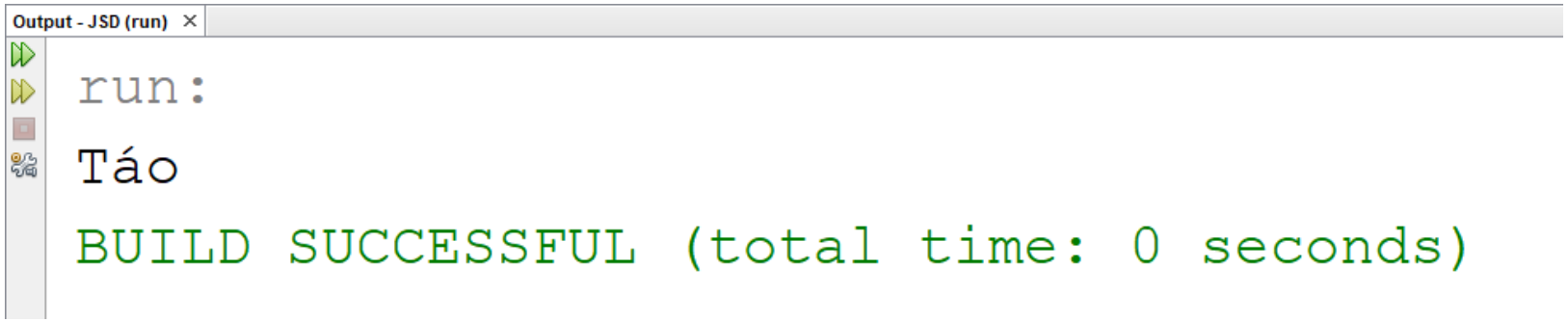
- Methods:
  - **put**(key, value): adds a (key, value) pair to the map. If the key already exists, updates its associated value.
  - **get**(key): accesses a value in the map with key.
  - **keySet**(): returns a Set of map's keys in unexpected order.
  - **values**(): returns a Collection of map's values in unexpected order.
  - **remove**(key): removes a (key, value) pair from map.
  - **containsKey**(key): returns true if the key exists in the map.

HashMap example 1

```
import java.util.HashMap;

public class MapDemo {

    public static void main(String[] args) {
        HashMap<String, String> m = new HashMap();
        m.put("Apple", "Táo");
        m.put("Cat", "Mèo");
        System.out.println(m.get("Apple"));
    }
}
```



HashMap example 2

```
HashMap<String, String> capitals = new HashMap<>();
capitals.put("England", "London");
capitals.put("Germany", "Berlin");
capitals.put("Norway", "Oslo");
capitals.put("USA", "Washington DC");
// prints London
System.out.println(capitals.get("England"));
```

HashMap example 3

```
HashMap<String, Integer> wordFreqs = new HashMap<>();
String[] arr = {"con", "ngua", "da", "da", "con", "ngua", "da"};
for (String s : arr) {
    if (wordFreqs.containsKey(s)) {
        int count = wordFreqs.get(s);
        wordFreqs.put(s, count + 1);
    } else {
        wordFreqs.put(s, 1);
    }
}
System.out.println(wordFreqs);
```

Program output:

```
{con=2, ngua=2, da=3}
```

Getting HashMap keys

```
HashMap<String, String> capitals = new HashMap<>();
capitals.put("England", "London");
capitals.put("Germany", "Berlin");
capitals.put("Norway", "Oslo");
capitals.put("USA", "Washington DC");
for (String s : capitals.keySet()) {
    System.out.println(s);
}
```

Program output:

```
USA
Norway
England
Germany
```

The order of keys is unexpected.

Getting HashMap values

```
HashMap<String, String> capitals = new HashMap<>();
capitals.put("England", "London");
capitals.put("Germany", "Berlin");
capitals.put("Norway", "Oslo");
capitals.put("USA", "Washington DC");
for (String s : capitals.values()) {
    System.out.println(s);
}
```

Program output:

```
Washington DC
Oslo
London
Berlin
```

The order of values is also unexpected.

`java.util.TreeMap`

- Similar public interface as `HashMap`.
  - Difference in internal implementation.
- The keys are stored in a unique, ordered Set.

Bulk Operations on Collections

- Available on *most* `Collection` classes.
- Methods:
  - `boolean` **`addAll`**`(Collection c)`: adds all items from `Collection c` to this `Collection`.
  - `void` **`putAll`**`(Map m)`: similar to `addAll()`, for `Map`.
  - `boolean` **`containsAll`**`(Collection c)`: check if all items of `Collection c` exists in this `Collection`.
  - `boolean` **`removeAll`**`(Collection c)`: removes all items in `Collection c` from this `Collection`.

Bulk Operations on Collections (cont.)

- `Object[] toArray()`: returns an array of all items.
- `T[] toArray(T[] a)`: returns an array of all items. The returned array has the same type as the input parameter's.
- `void clear()`: empty this `Collection`.

- Difference between `toArray()` and `toArray(T[] a)`:

```
LinkedList<String> source = new LinkedList<String>();
// Fill in some data
Object[] array1 = source.toArray();
String[] array2 = source.toArray(new String[0]);
```

The `java.util.Arrays` class

- Provides static methods for manipulating arrays.
  - `binarySearch()`: searching sorted arrays
  - `equals()`: comparing arrays
  - `fill()`: placing values into arrays (can fill either all or part of the array)
  - `sort()`: sorting arrays (can sort all or part of the array)
- Methods are overloaded to work with primitive-type arrays and reference-type arrays.
- `System.arraycopy()`: copy a portion of one array into another.

Demo using java.util.Arrays

```
import java.util.Arrays;

public class UsingArrays {
    public static void main(String[] args) {
        int intValues[] = {1, 2, 3, 4, 5, 6};
        double doubleValues[] = {8.4, 9.3, 0.2, 7.9, 3.4};
        int filledInt[], intValuesCopy[];
        filledInt = new int[10];
        intValuesCopy = new int[intValues.length];

        // fill with 7s
        Arrays.fill(filledInt, 7);
        // sort doubleValues ascending
        Arrays.sort(doubleValues);
        // copy array intValues into array intValuesCopy
        System.arraycopy(intValues, 0, intValuesCopy,
                0, intValues.length);
    }
}
```

Convert an array into a List

```
String suits[] = {"Hearts", "Diamonds",
        "Clubs", "Spades"};
List list = new ArrayList(Arrays.asList(suits));
List list2 = Arrays.asList(suits);
list2.set(0, "AAA");
System.out.println(Arrays.toString(suits));
```

- `list` is independent of `suits`, changes to either does not affect the other.
- `list2` is a "view" of `suits`, changes made to `list2` changes `suits` and vice versa.