COLLECTIONS

Programming Principles II

Topics

Collections Framework

Problem

By now, we can store data as primitive types, Objects and arrays. In some cases, we need some more functionality to manipulate data, like:

- Having array always in a sorted order
- Keep only unique objects in our array
- Add elements to the start or end of the array
- Keep some identifiers for the array elements, like Student ID
- Find object in an array according to its value (how to find a student object in an array that has firstName="Samir"?)

java.util

java.util package provides variety of functions concerning date and time, formatting, number and string operations, bit manipulations and so on.

Starting JDK9, java.util. is a part of the ***java.base*** module.

Collections is one of the powerful functionality provided by **java.util**.

Collections consists of many interfaces and classes.

Collections Framework

Collections Framework standardizes the way in which groups of objects are handled by Java programs.

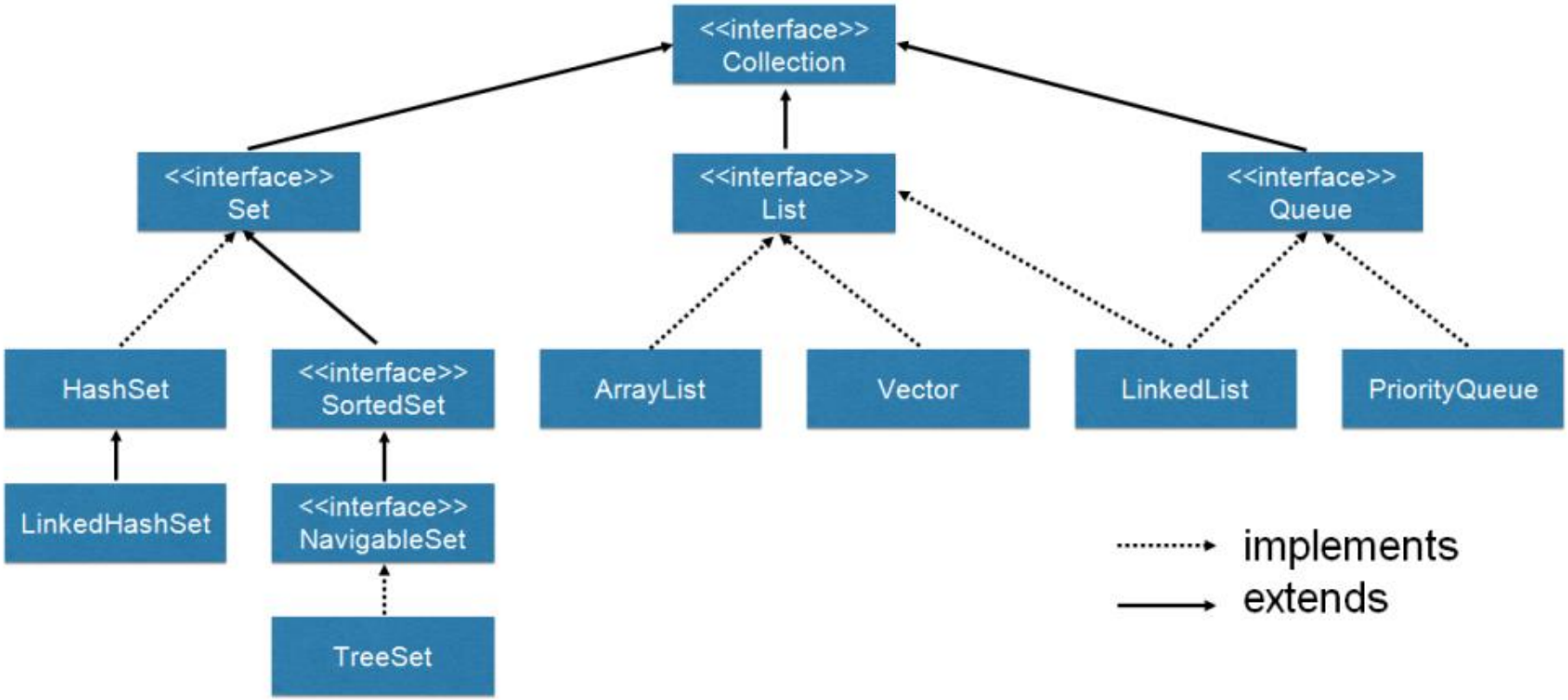
The framework brings several possibilities:

- speed of operation on arrays and data sets

- easiness of implementation through the similar signature

- extension of the existing collections

Interfaces



Source: https://marcus-biel.com/java-collections-framework/

Topics

Collections Interfaces

Collection

INTRODUCTION & INTERFACES

Interfaces – *Collection*

The **Collection** interface is the foundation upon which the *Collections Framework* is built and declares the core methods that all collections will have:

Method	Description
boolean add(Object o)	insert an element into collection
boolean addAll(Collection c)	insert a collection into collection
void clear()	remove all elements from collection
boolean remove(Object o)	delete an element from collection
boolean removeAll(Collection c)	delete all from specified collection
boolean retainAll(Collection c)	remove all elements except collection c
boolean contains(Object o)	used to search an element
boolean containsAll(Collection c)	used to search an collection
boolean equals(Object o)	used to check quality of object
boolean isEmpty()	check the collection is empty or not
int size()	get number of element in a collection
int hashCode()	return the hashcode number for collection
Iterator iterator()	return an iterator
Object[] toArray()	convert the collection to array

Source: https://www.javafoundation.xyz/2016/06/collection-interface.html

Interfaces – *List*

The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

A list ***may contain duplicate elements***.

Additional to the Collection method, Lists introduces its own methods. Some of them:

Method	Explanation
E **get**(int index)	Returns the element at the specified position in this list.
E **set**(int index, E element)	Replaces the element at the specified position in this list with the specified element
int **indexOf**(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

Interfaces – *Set*

The **Set** interface defines a set. It extends **Collection** and specifies the behavior of a collection that ***does not allow duplicate elements***.

Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set.

List (duplicate allowed)		**Set** (duplicates are not allowed)	
5	“Aslan”	5	“Aslan”
1	“Ceyhun”	1	“Ceyhun”
2	“Samir”	2	“Samir”
3	“Anar”	3	“Anar”
4	“Anar”	4	“Sahib”
2	“Samir”	8	“Zakir”
5	“Hilal”	6	“Hilal”

Interfaces – *SortedSet*

The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order.

In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized in below table:

Method	Explanation
E **first**()	Returns the first (lowest) element currently in this set.
E **last**()	Returns the last (highest) element currently in this set.
SortedSet **subSet**(E start, E end)	Returns a view of the portion of this set whose elements range from start, inclusive, to end, exclusive
SortedSet **tailSet**(E start)	Returns a view of the portion of this set whose elements are greater than or equal to *start*.

Interfaces – *NavigableSet*

he **NavigableSet** interface extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

In addition to those methods provided by **SortedSet**, the **NavigableSet** interface declares additional methods like:

Method	Explanation
E **ceiling**(E e)	Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
E **floor**(E e)	Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
E **higher**(E e)	Returns the least element in this set strictly greater than the given element, or null if there is no such element.

Interfaces – *Queue*

The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a ***first-in, first-out list***. A *NullPointerException* is thrown if an attempt is made to store a null object since null elements are not allowed in a queue. It also introduces additional methods:

Method	Explanation
boolean **offer**(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E **element**()	Retrieves, but does not remove, the head of this queue.
E **remove**()	Retrieves and removes the head of this queue.
E **peek**()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E **poll**()	Retrieves and removes the head of this queue or returns null if this queue is empty.

Interfaces – *Queue*

- Queue elements can only be removed from the head of the queue.
- There are two methods that obtain and remove elements: **poll( )** and **remove( )**. The difference between them is that **poll( )** returns **null** if the queue is empty but **remove( )** throws an exception.
- There are two methods, **element( )** and **peek( )**, that obtain but don't remove the element at the head of the queue. They differ only in that **element( )** throws an exception if the queue is empty, but **peek( )** returns **null**.
- **offer( )** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, **offer( )** can fail.

Interfaces – *Queue*

	Throws exception	Special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Obtain	get()	peek()

Interfaces – *Deque*

The ***Deque*** interface extends *Queue* and declares the behavior of a double-ended queue. Double-ended queues can function as standard, ***first-in, first-out*** queues or as ***last-in, first-out*** stacks.

A *NullPointerException* is thrown if an attempt is made to store a null object since null elements are not allowed in the deque.

- Deque includes the methods ***push()*** and ***pop()***. These methods enable a Deque to function as a stack.
- ***addFirst()*** and ***offerFirst()*** methods allow to add elements to the start of deque.
- ***addLast()*** and ***offerLast()*** methods allow to add elements to the end of the deque.

Interfaces – *Deque*

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Obtain	getFirst()	peekFirst()	getLast()	peekLast()

Questions

Topics

Collections Classes - Lists

Classes - ArrayList

The **ArrayList** class extends **AbstractList** and implements the **List** interface.

In Java, standard arrays are of a fixed length. *ArrayList* supports dynamic arrays that can grow as needed. If the elements are not known, an ArrayList can be created using an empty constructor:

```
ArrayList al1 = new ArrayList();

List al2 = new ArrayList();
```

The elements are added to the array using *add()* method. All other methods of the Collection and List interfaces are applicable as well:

```
al.add("Hello");
al.add("World");
al.remove("Hello");
al.set(0,"New");
```

*Running ahead - <E> notation

The below notation you might see in tutorials

```
ArrayList<E> al = new ArrayList<E>();
```

means the elements of the ArrayList can be only of **E type**.

Example:

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

will restrict adding any object to the list except Strings.

This topic will be covered in Generics section

ArrayList - example

```
import java.util.*;

class ArrayListDemo{
	public static void main(String args[]) {
		ArrayList al = new ArrayList();
		al.add("A");
		al.add("B");
		al.add("C");
		al.add("D");
		al.add("E");
		System.out.println("Array size: "+al.size());

		al.remove("C");
		al.remove(2);
		al.set(0,"X");
		System.out.println("Array size: "+al.size());

		for (int i=0; i<al.size(); i++)
		   System.out.println(al.get(i));
		}
}
```

Retrieving an array

toArray() methods allows to get the contents of the ArrayList as a Java array:

```
Object [] toArray();
```

Getting a Java array might be useful when speed is a priority, or some class method accepts only an array.

Example:

```
String [] strArr = al.toArray(new String[al.size()]);
```

(works with Generics)

Read more: https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods/

Classes - LinkedList

The ***LinkedList*** class extends *AbstractSequentialList* and implements the *List*, *Deque* (which in turn extends Queue) interfaces.

Since *LinkedList* implements the *Deque* interface, you have access to the methods defined by Deque.

```
LinkedList ll = new LinkedList();

ll.add("A");
ll.add("B");
ll.add("C");
ll.addLast("Y");
ll.addFirst("X");
System.out.println("The last element is: " +
                                        ll.getLast());
for (Object s : list) {
    System.out.println(s);
}
```

Questions

Topics

Iterators

Iterator

The **Iterator** interface enables the navigation through a collection, getting or removing elements.

Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection.

```
ArrayList al = new ArrayList();

…

Iterator it = al.iterator();
```

The **hasNext()** method check if there is a next element in the collection:

```
boolean bNext = it.hasNext();
```

Iterator

The **next()** method retrieves the next element of the collection and moves cursor to that position:

```
Object obj = it.next();
```

The **remove()** method removes the last element returned by the iterator:

```
it.remove();
```

This method can be called only once per call to **next()**.

Iterator - example

```
ArrayList al = new ArrayList();
al.add(”One”);
al.add(“Two”);
al.add(“Three”);

Iterator it = al.iterator();
while (it.hasNext()) {
   String s = (String) it.next();
   System.out.println(“Next element: ”+s);
   }
```

ListIterator - example

```
ArrayList al = new ArrayList();
al.add(”One”);
al.add(“Two”);
al.add(“Three”);

ListIterator it =
                list.listIterator(list.size());
while (it.hasPrevious()) {
   String s = (String) it.previous();
   System.out.println(“Next element: ”+s);
   }
```

Note: In this example, calling hasNext() would return false.

But if you check for hasPrevious() but still call it.next(), what would happen?

For-each alternative

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the **for** loop is often a more convenient alternative to cycling through a collection than is using an iterator:

```
ArrayList al = new ArrayList();
…

for (Object o : al) {
   System.out.println("Next element: " + o);
   }
```

Questions

Topics

Collection classes - Sets

HashSet

The ***HashSet*** class extends ***AbstractSet*** and implements the ***Set*** interface.

```
HashSet set1 = new HashSet();

Set set2 = new HashSet();
```

or it is possible to create a HashSet with an initial capacity (number of elements):

```
HashSet hSet = new HashSet(10);
```

Hash set **does not guarantee the order** of its elements. The elements might be inserted in one order but read in another.

HashSet – Example 1

```
HashSet hs = new HashSet();
    hs.add(2);
    hs.add(3);
    hs.add(5);
    hs.add(3);
    hs.add(2);
    hs.add(3);
    hs.add(5);
System.out.println(hs); // will print as [2, 3, 5]
```

HashSet – Example 2

```
HashSet hs = new HashSet();
    hs.add(2);
    hs.add(3);
    ….
    Iterator it = hs.iterator();
    while (it.hasNext()) {
       System.out.print(it.next() + ", ");
    }
    // will print as 2, 3, 5, …
```

LinkedHashSet

LinkedHashSet extends *HashSet* and maintains a list of the elements in the order in which they were inserted. This allows insertion-order iteration over the set.

That is, when iterating through a collection-view of a LinkedHashSet, the elements will be returned in the order in which they were inserted.

It achieves that by maintaining a doubly-linked list running through its elements.

The order of an element is not affected if that element is re-inserted.

TreeSet

The ***TreeSet*** class extends *AbstractSet* and implements the *NavigableSet* interface.

A TreeSet provides an efficient means of storing values in sorted order and allows rapid data retrieval. Unlike a hash set, a tree set guarantees that its elements will be sorted in ascending order of values.

```
TreeSet ts = new TreeSet();
```

The elements are sorted based on their natural ordering.

TreeSet – Example 1

```
Set ts = new TreeSet<>();
        ts.add(2); ts.add(5);
        ts.add(3); ts.add(10);
        ts.add(1); ts.add(5);
        ts.add(2); ts.add(3);

Iterator it = ts.iterator();
while (it.hasNext()) {  System.out.print(it.next() + ", ");  }
// will print as 1, 2, 3, 5, 10,
```

TreeSet – Example 2

```
TreeSet ts = new TreeSet<>();
        ts.add(2); ts.add(5);
        ts.add(3); ts.add(10);
        ts.add(1); ts.add(5);
        ts.add(2); ts.add(3);

Iterator it = ts.descendingIterator();
while (it.hasNext()) {  System.out.print(it.next() + ", ");  }
// will print as 10, 5, 3, 2, 1,
```

Questions

Topics

Map Interfaces

Maps

A *map* is an object that stores associations between keys and values, or *key/value pairs*. Both keys and values shall be objects:

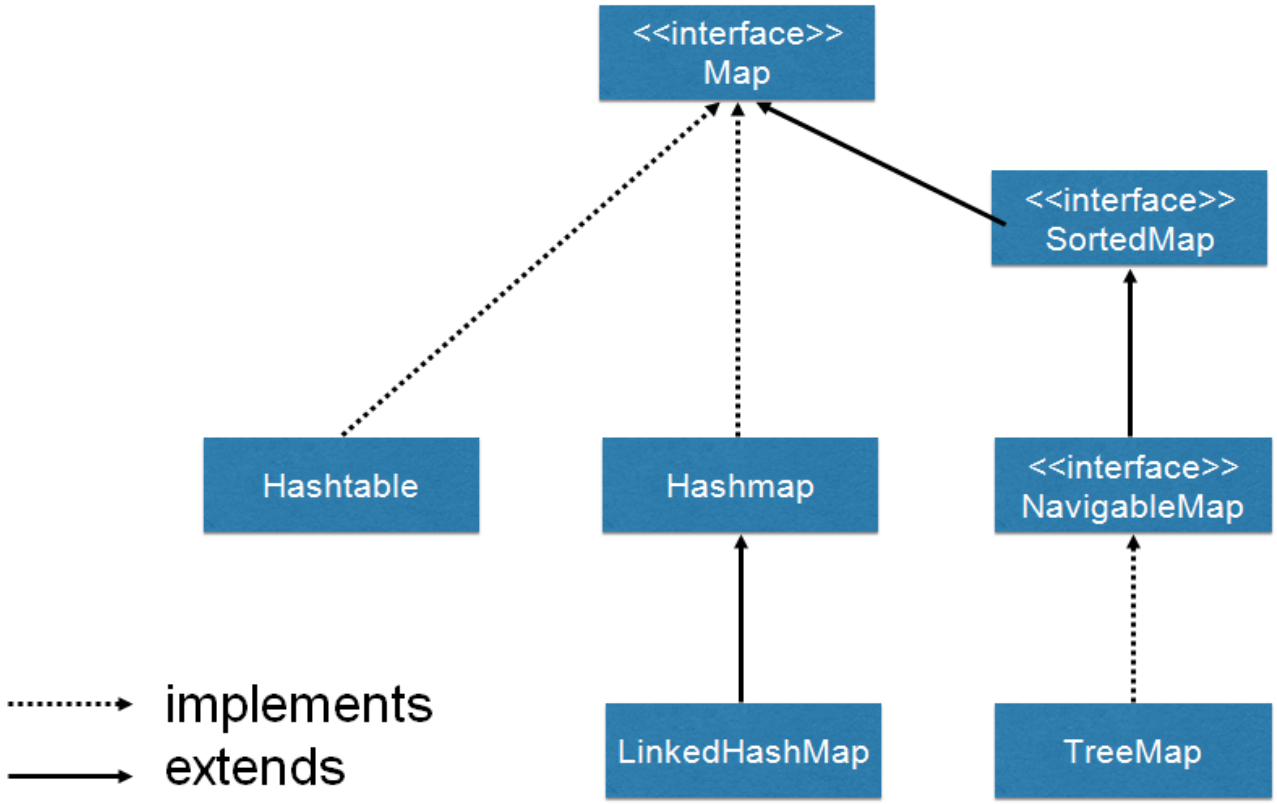
```
("SPIA", "School of Public and International Affairs")
              ("SB", "School of Business")
        ("SITE", "School of IT and Engineering")
```

A value can be found by the provided key. ***Keys in map are unique***.

Maps are considered as a part of the Collections Framework but ***do not inherit*** the **Collection** interface.

Read more: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Map.html

Interfaces



Source: https://marcus-biel.com/java-collections-framework/

Maps

Maps revolve around two basic operations: **get( )** and **put( ):**

- To put a value into a map, use **put(key, value)**
- To obtain a value, call **get(key)**

put(oldKey,newValue) will replace the old value (does not add a new pair)

There is no iterator() method in Maps. As an alternative, there is a keySet() method that returns the set of keys, which can be iterated:

```
Set keys = someMap.keySet();
Iterator it = keys.iterator();

while (it.hasNext()) {
   Object key = it.next();
   Object value = someMap.get(key);
   }
```

Some Map interfaces

The ***SortedMap*** interface extends **Map**. It ensures that the entries are maintained in ascending order based on the keys. It introduces useful methods like *firstKey()* and *lastKey()* that help take the very first and last key of the map, correspondingly.

The ***SortedMap*** also has methods for selection of the subsets of the map: *subMap(start,end)*, *headMap(end)* and *tailMap(start)*.

The ***NavigableMap*** interface extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.

Topics

Map Classes

HashMap

The HashMap class **extends** AbstractMap and **implements** the Map interface. There is no additional method introduced by HashMap itself:

```
HashMap hm = new HashMap();
```

or it is possible to create a HashMap with an initial capacity (number of elements):

```
HashMap hm = new HashMap(10);
```

Hash map **does not guarantee the order** of its elements. The orders might be inserted in one order but read in another.

HashMap – Example 1

```
HashMap hm  = new HashMap();

hm.put(1, “Monday”);
hm.put(2, “Tuesday”);
hm.put(3, “Wednesday”);
hm.put(4, “Thursday”);
hm.put(5, “Friday”);
hm.put(6, “Saturday”);
hm.put(7, “Sunday”);

System.out.println(hm); // will print as {1=Monday, 2=Tuesday, …}
```

HashMap – Example 2

```
HashMap hm  = new HashMap();

hm.put(1, “Monday”);
hm.put(2, “Tuesday”);
…
Set keys = hm.keySet();
Iterator it = keys.iterator();

while (it.hasNext()) {
   Object key = it.next();
   String value = (String) hm.get(key);
   System.out.println(“key=“+key+” value=”+value);
   }
```

LinkedHashMap

LinkedHashMap extends HashMap and maintains a list of the entries in the order in which they were inserted. This allows insertion-order iteration over the map.

That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.

It achieves that by maintaining a doubly-linked list running through its entries (keys-value pairs).

The order of entries is not affected if a key is re-inserted into map.

Tree Map

The ***TreeMap*** class **extends** AbstractMap and **implements** the NavigableMap interface.

A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. Unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

```
TreeMap tm = new TreeMap();
```

TreeMap does not introduce any additional method – it just implements the methods of interface and the *AbstractMap* class

HashMap / LinkedHashMap / TreeMap

{“B”, “Brown”}
{“W”, “White”}
{“G”, “Green”}
{“P”, “Purple”}
{“Y”, “Yellow”}
{“R”, “Red”}

{key,value} inserted to the Map

HashMap (random order)

{“G”, “Green”}
{“W”, “White”}
{“B”, “Brown”}
{“Y”, “Yellow”}
{“R”, “Red”}
{“P”, “Purple”}

LinkedHashMap (inserted order)

{“B”, “Brown”}
{“W”, “White”}
{“G”, “Green”}
{“P”, “Purple”}
{“Y”, “Yellow”}
{“R”, “Red”}

TreeMap (natural order)

{“B”, “Brown”}
{“G”, “Green”}
{“P”, “Purple”}
{“R”, “Red”}
{“Y”, “Yellow”}
{“W”, “White”}

Hands On Exercise

Given a file with a large text in it. Count each distinct word and print N most frequently used word in the text.

Can we do the same for **k-word** pieces?

Homework

Write a program that checks username / password using Maps. Think, what Map implementation will you use?

Take login and password as arguments like

```
java MyAuthClass login password
```

or using java.util.Scanner:

```
Scanner scanner = new Scanner(System.in);
String login = scanner.nextLine();
String password = scanner.nextLine();
```

Questions

Topics

Some Utility classes

Arrays

The ***java.util.Arrays*** class provides various static methods that are useful when working with arrays. Some of them:

Method (all static)	Explanation
void sort(type [] arr)	sorts an array in ascending order.
int binarySearch(type [] arr, type val)	Performs a binary search to find a specified value. This method must be applied to sorted arrays
type[ ] copyOf(type[ ] source, int len)	returns a copy of an array with a given length from the start. "If the copy is longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays).
boolean equals(type [] a1, type [] a2)	returns true if two arrays are equivalent.
void fill(type [] arr, type value)	assigns a value to all elements in an array.

Read more: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html

Arrays.sort

```
import java.util.*;

public class ArrDemo {
   public static void main(String[] args) {
     int [] intArr = { 5,-1,2,3,55,3,-5};
     String [] strArr = { "Konul", "Sariyya", "Aydin", "Habil", "Natavan"};
     Arrays.sort(intArr);
     Arrays.sort(strArr);

     for (int i : intArr)
       System.out.print(i + " ");
     System.out.println();

     for (String s : strArr)
       System.out.print(s + " ");

     }
}
```

Arrays.binarySearch

If the input array is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

The method returns the index of the search key, if it is contained in the array; otherwise, (-(*insertion point*) - 1). (read more in Java docs about the insertion point).

```
import java.util.*;

public class ArrDemo {
    public static void main(String[] args) {
        int [] intArr = { 5,-1,2,3,55,3,-5};
        int idx = Arrays.binarySearch(intArr,-5);
        System.out.println("Search result without sorting: "+idx);

        Arrays.sort(intArr);
        idx = Arrays.binarySearch(intArr,-5);
        System.out.println("Search result with sorting: "+idx);
    }
}
```

Arrays.fill

fill() method is useful when you want to have some initial values of the array elements other than 0 or null. This method works with one-dimensional arrays. The following example demonstrates a creation of 2-dimensional array of 1s:

```
import java.util.*;

public class ArrDemo {
    public static void main(String[] args) {
        int [][] matrixOfOnes = new int[5][5];

        for (int[] row: matrixOfOnes)
            Arrays.fill(row,1);

        for (int i=0;i<matrixOfOnes.length;i++) {
            System.out.print("| ");
            for (int j=0;j<matrixOfOnes[i].length;j++)
                System.out.print(matrixOfOnes[i][j]+" ");
            System.out.println("|");
        }
    }
}
```

Arrays.mismatch

int mismatch(type [] arr1, type[] arr2) finds and returns the index of *the first mismatch* between two arrays, otherwise return -1 if no mismatch is found.

```
import java.util.*;

public class ArrDemo {
  public static void main(String[] args) {
    int [] intArr1 = { 5,-1,2,3,55,3,-5};
    int [] intArr2 = { 5,-1,2,7,53,3,5};

    int idx = Arrays.mismatch(intArr1,intArr2);
    System.out.println("Mismatch index: "+idx);
    System.out.println("Elements "+intArr1[idx]+" and
                        "+intArr2[idx]+" do not match");
    }
}
```

Collections

The ***java.util.Collections*** class provides various static methods that are useful when working with collections. Some of them:

Method (all static)	Explanation
<T extends Comparable<? Super T>> void sort(List<T> list)	Sorts the specified list into ascending order, according to the natural ordering of its elements.
<T> int binarySearch(List<? Extends Comparable<? super T>> list, T key)	Performs a binary search to find a specified value. This method must be applied to sorted lists
<T> List<T> nCopies(int n, T o)	Returns an immutable list consisting of n copies of the specified object.
<T> void fill(List<? super T> list, T obj)	Replaces all of the elements of the specified list with the specified element.
void shuffle(List<?> list)	Randomly permutes the specified list using a default source of randomness.

Read more: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html

StringTokenizer

In practice we need to split a text into parts by a given delimiter, like getting words of the text, or reading an elements of the comma-delimited structure.

StringTokenizer helps realize the mentioned task:

StringTokenizer(String *str*)
StringTokenizer(String *str*, String *delimiters*)

- The first constructor accepts a string and fetches parts(tokens) by space delimiter.
- The second constructor sets the delimiter symbol. It can be a single symbol like "," or set of delimiters like ",#_".

StringTokenizer(str, ",#_") will split text into tokens if one of the given delimiters are met. Delimiter characters themselves will not be treated as tokens (test it!).

StringTokenizer

Once a **StringTokenizer** object created, the **nextToken( )** method is used to extract consecutive tokens.

The **hasMoreTokens( )** method returns **true** while there are more tokens to be extracted:

```
StringTokenizer st = new StringTokenizer("Hello World today is Monday");

while (st.hasMoreTokens()) {
    String word = st.nextToken();
    System.out.println(word);
    }
```

Optional

The object returned by methods might be null and this may lead to an exception or unwanted result:

```
String text = startsWith("A");     // the method may return null
System.out.println(text.toUpperCase());
```

Beginning with JDK 8, the class called ***Optional*** (and also ***OptionalDouble***, ***OptionalInt***, and ***OptionalLong***) offer a new way to handle situations in which a value may or may not be present.

The methods that may return no data, can return an Optional object:

```
Optional<String> value = startsWith("A");
```

The content of the result can be tested with *isPresent()* and the value is fetched with *get()* method:

```
if (value.isPresent())
    System.out.println(value.get());
```

Optional

The Optional object is created using its static *of(type)* method:

```
Optional value = Optional.of("Hello");
```

Example:

```
Optional returnElem(String [] arr, int idx) {
    Optional retval = Optional.empty();

    if (idx <= arr.length)
        retval = Optional.of(arr[idx]);
    return retval;
    }
```

Optional vs returning Null

Why can't we just use

```
String text = startsWith("A");
if (text!=null)
   System.out.println(text.toUpperCase());
```

And the counter-question is "what if you forget about that case?"

The simplest reason to return Optional is to force the user (or yourself) to consider the case that there can be no value.

Interesting article on Optional: https://dzone.com/articles/using-optional-correctly-is-not-optional

Questions