COMP122 Week 8

Design Patterns

UNIVERSITY OF LIVERPOOL

Dr. Patrick Totzke
totzke@liverpool.ac.uk

https://liverpool.instructure.com/courses/59716

Collections

Collections

A Collection (also Aggregate) is a box that stores objects.

- Collections are objects
- they can be dynamically resized.

Aggregates come in all forms and shapes and differ for instance by

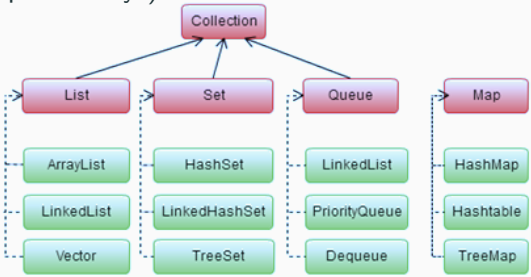
- what type of elements they store
- how to access its contents.
- how they store it (as list, tree,...)

Collections in Java

Java comes with a bunch of ready to use **interfaces**:

- `Collection` for a group of objects.
- `Queue` for a FIFO or LIFO data structure.
- `Set` for a collection with no duplicate elements.
- `List` for an ordered collection (duplicates allowed).
- `Map` which maps keys to values (no duplicate keys)

... and **implementations**:



Syntax by Example

This is how you'd create a collection that holds a list of strings.

```
1  import java.util.*          // the Collections framework lives here
2  List<String> customers;              // declare a list of strings
3  customers = new ArrayList<String>();  // instantiate one
```

Comments

- `List<String>` is the identifier of the interface.
- `ArrayList<String>` is its implementation, i.e., a concrete data type

Side Note

the special syntax is an example of java generics, or parametrized types. This is to avoid having to define collection interfaces/classes for every element-type individually.

ArrayList Example

```java
import java.util.ArrayList;

public class ArrayListUser {

    public static void main(String[] args){
        ArrayList<Integer> numbers = new ArrayList<Integer>();

        for (int i = 1; i < 100; i++){
            numbers.add(i*i);
        }

        System.out.println( numbers.get(34) );  // prints 1225 = 35*35
        numbers.remove(34);                     // remove an element
        System.out.println( numbers.get(34) );  // access the same index
    }
}
```

```
$ javac .java && java ArrayListUser
1225
1296
```

ArrayList Example – Comments

- Collections store objects, and **not** primitive data types. This is why we use the wrapper class `Integer` instead of `int` here.
- Notice that `ArrayList` re-indexes its elements after one is removed.
- `Collection` offers a `size()` method to read off the current number of elements. For example, try adding the following line to our program.

```
1  System.out.println( numbers.size() );
```

- You can declare an initial size when instantiating an ArrayList

```
1  List<Integer> numbers = new ArrayList<Integer>(25);
```

Some Other Collections

Other useful types from `java.util` are `Stack` and `Queue`, as well as `Map`s. These are common data structures used in algorithmic tasks (searching graphs etc.)

A Stack

can only have elements added to, or removed from, the end of the list. Hence, a `Stack` is referred to as a "last-in/first-out" or LIFO structure. `java.util.Stack` is a class.

A Queue

allows insertions at the end but removals only from the front, so is referred to as "first-in/first-out" (FIFO) data structure. `java.util.Queue` is an Interface and has several implementations, e.g. `PriorityQueue`.

A Map

is a data structure that stores (key, value) pairs. `java.util.Map` is an interface. We'll see an example in a minute.

Some Other Collections – Stack example

Creating one of these other collections is similar to what we have seen.

```
1  import java.util.Stack;
2
3  public class StackUser {
4
5     public static void main (String args[]) {
6        Stack<Integer> st = new Stack<Integer>();
7
8        st.push(12);  // push an element to the stack
9        st.push(-1);
10       st.push(10);
11       System.out.println(st);
12
13       int x = st.pop();  // pop the topmost stack element
14    }
15 }
```

Example: Word Counts

Suppose that we're building an application to analyse a text file and print, for each word, the number of times it occurs. We need to

1. Open the file for input. (week 10)
2. Read the file line by line (week 10)
3. Break the text into individual words.
4. Keep track of the word count for each individual word.

Maps

- Recall that a `Map` is a data structure to store (key, value) pairs.
- A natural choice for storing word counts would be a map where keys are `String`s and values are `Integer`s.
- We will use `java.util.HashMap`, which implements the interface `Map`.

```
1  Map<String, Integer> wc = new HashMap<String, Integer>();
```

- Note: `HashMap<K,V>` requires that `K.hashCode()` exists.

HashMap Demo

```java
// create a mapping from Strings to integers.
Map<String, Integer> wc = new HashMap<String, Integer>();

// store a (key, value) pair
wc.put("place", 3);

// access the value associated with a key
int number = wc.get("place");

// check if key has a value
bool contains = wc.containsKey("and");

// remove some pair completely
wc.remove("the");

// iterate over all keys
for ( String k : wc.keySet() ) {
    System.out.println("key: " + k + " value: " + wc.get(k));
}
```

Example: Word Counts

```
1  String[] wordlist = ... ;  // a given list of words (Strings)
2
3  HashMap<String,Integer> wordCount;  //  list of (word, count) pairs
4  wordCount = new HashMap<String,Integer>();  //  empty HashMap
5
6  for (String w : wordlist){
7    if (!w.isEmpty()) {
8      if (wordCount.containsKey(w)) {
9        wordCount.put(w, wordCount.get(w) + 1);
10     }
11     else {
12       wordCount.put(w, 1);
13     }
14   }
15 }
16
17 wordCount.size();  // the number of different words
18 wordCount.get("and");  // the number of times "and" occurs
19 // See FileWordCount.java for a fully functional program
```

Iterators

Iterators – Motivation

Can you see the problem with the following code?

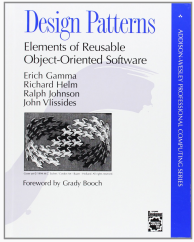
```
1  // numbers is the initialized ArrayList from before.
2  int size = numbers.size();
3  for (int i; i<size; i++) {
4    int j = numbers.get(i) + 1;  // do stuff with the current number
5    numbers.remove(i);  // remove it from the list
6  }
```

Removing a list element (line 5) changes the size of the List.
The termination condition depends on the size before the first removal and so line 4 will eventually try to access an element at a too-large index!

Demo? (see `Unsafe.java`)

Iterators

...provide a way to access elements of an aggregate object sequentially without exposing its underlying representation.



- Iterators are objects
- single-use (create a new one every time you iterate over the collection)
- Separating the traversal mechanism from the collection allows for different traversal policies without cluttering the collection class.
- can provide safe access to their underlying collection
- are often created by the collection itself

Iterators – Example

Instead of iterate over out list with a `for` loop

```
1  for (int i; i<numbers.size(); i++) { ... }
```

or

```
2  for (int i : numbers)  { ... }
```

we can an `Iterator` like this

```
3  Iterator<Integer>  iter = numbers.iterator();
4
5  while ( iter.hasNext() ) {
6    int i = iter.next();
7    int j = i*i;
8  }
```

Iterators (cont.)

- You must specify the type of object that the iterator will be getting from the collection (e.g. `Integer` in the previous example).
- The `Iterator` interface implements a safe `remove()` method that removes the last element returned by the `Iterator.next()` method. This method can be used to remove some (or all) of the elements while using the `Iterator` to process the list.
- This method is different from the `remove()` method of the collection.

Iterators – Safe Removal (I)

```
1  import java.util.*;
2
3  public class Safe {
4
5      public static void main(String[] args) {
6
7          // create a list of Strings
8          List<String> list = new ArrayList<>();
9
10         // add stuff
11         list.add("A");
12         list.add("B");
13         list.add("C");
14
15         // print content. This should print "[A, B, C]"
16         System.out.println(list);
17
```

Iterators – Safe Removal (II)

```
18          // get an iterator from the collection
19          Iterator<String> iterator = list.iterator();
20
21          while(iterator.hasNext()) {
22              String value = iterator.next();  // current element
23
24              if(value.equals("B")){   // Remove element
25                  iterator.remove();
26              }
27          }
28          System.out.println(list);
29      }
30  }
```

```
$ javac SafeIt.java && java SafeIt
[A, B, C]
[A, C]
```

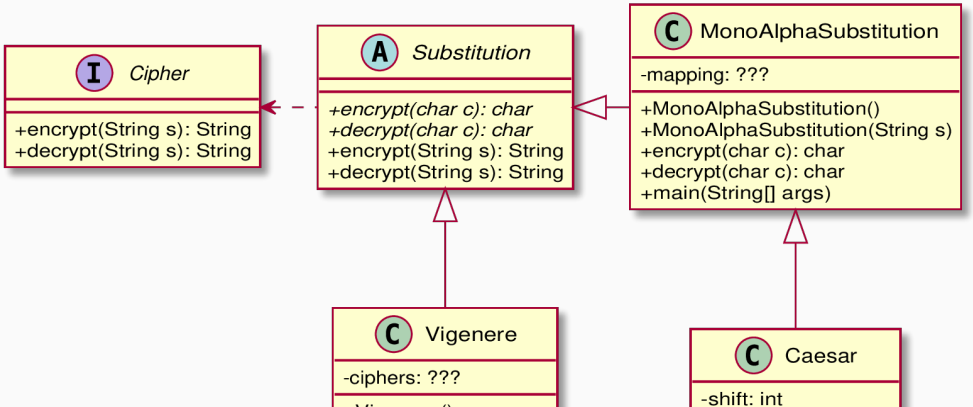
Final remarks on Collections

- Java Collections provide a natural way of dealing with a group of (similar) objects. The interfaces are set up to give common syntax for inserting, deleting, and searching for elements in a collection.
- All related classes/interfaces are defined in `java.utils`.
- You can also utilize polymorphism in a collection: For instance, and `ArrayList<Ship>` object can hold elements of type `Ferry` and `Liner`.
- use `Iterator`s to iterate over, and safely modify collections

Your Questions

Q44

I'm really confused as to what were meant to do in part 1 of the assignment. Like am I meant to just the return the abstract class with no actual functions coded just like we been doing in lbs last week or do I fully code the encryption and decryption process similar to the first one.



Summary

We looked at...

- Java Collections: `List`, `Queue`, `Stack`, `Maps`
- Iterators

Remember

- **Assignment 2 is due this Friday 5pm**
- Week 9 starts April 17

Next up, in Week 9

- Exceptions/Errors
- Exception Handling in Java