OOP
Lab 9.

🎯 Objectives

- Interfaces, enums
- Polymorphism, overriding `equals`
- Singleton

Exercise 1

In this lab, we implement a **queue** data structure to store elements of type Object in two different ways (see Fig. 1).
A queue operates based on the principle of *First In First Out* (FIFO).

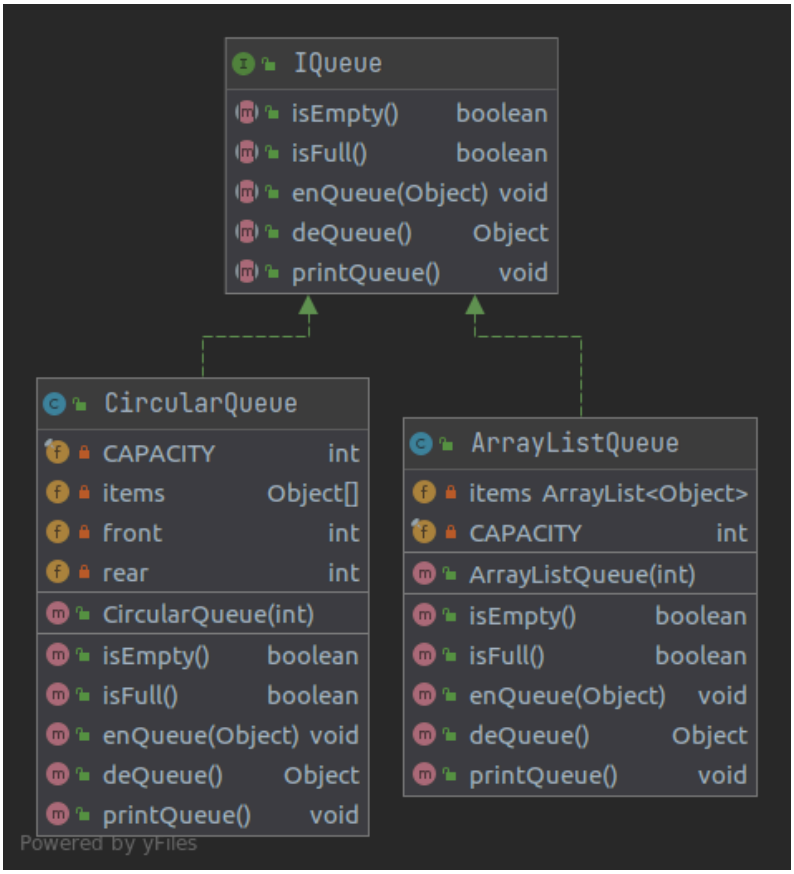

Fig. 1. Class diagram for queue implementations

OOP
Lab 9.

I. IQueue interface

First, we declare an `IQueue` interface which declares the common queue operations such as:

- `enQueue` - add a new element to the queue
- `deQueue` - removes the first element of the queue and returns it
- `printQueue` - prints the content of the queue
- `isEmpty`
- `isFull`

II. ArrayListQueue

The first `ArrayListQueue` class implements the `IQueue` interface and uses an `ArrayList` to store the items, and has a maximal capacity (CAPACITY).

III. CircularQueue

The second `CircularQueue` class implements the `IQueue` interface and uses a classical array for storing the items with size CAPACITY.

Useful link: https://www.programiz.com/dsa/circular-queue

IV. Main

Test your implementations (`ArrayListQueue` and `CircularQueue`) using the following code.

```
IQueue queue = new ArrayListQueue( 5 ); // new CircularQueue( 5 );
Random rnd = new Random();
for( int i=0; i<100; ++i ){
   int value = rnd.nextInt(100);
   if( value < 50 ){
       System.out.println("Add: " + i);
       queue.enQueue( i );

   } else{
```

```
        if( queue.isEmpty() ){
            System.out.println("Cannot delete from an empty queue");
        } else {
            int element = (Integer) queue.deQueue();
            System.out.println("Deleted: " + element);
        }
    }
    queue.printQueue();
}
```

V. equals()

Override the `equals` method for the class `ArrayListQueue`. Two queues are equal if and only if they contain the same items in the same order. Also, they must be instantiated from the same class. The capacities of the queues may be different.
The following code should print **true**!

```
IQueue q1 = new ArrayListQueue(5);
IQueue q2 = new ArrayListQueue(10);
for( int i=0; i<5; ++i){
   q1.enQueue( i );
   q2.enQueue( i );
}
System.out.println( q1.equals( q2 ));
```

Override the `equals` method for the class `CircularQueue`. Two queues are equal if and only if they contain the same items in the same order. The capacities of the queues must be the same.
The following code should print **true**!

```
IQueue q3 = new CircularQueue(5);
IQueue q4 = new CircularQueue(5);
for( int i=1; i<6; ++i){
   q3.enQueue( i );
}
q4.enQueue( 1 );
q4.enQueue( 1 );
for( int i=1; i<4; ++i){
   q4.enQueue( i);
```

```
}
q4.deQueue();
q4.deQueue();
q4.enQueue( 4 );
q4.enQueue( 5 );

System.out.println( q3.equals( q4 ));
```

Explanation:

```
q3:   CircularQueue [ array:[ 1 2 3 4 5 ], front: 0, rear: 4]
q4:   CircularQueue [ array:[ 4 5 1 2 3 ], front: 2, rear: 1]
```

Exercise 2

In this exercise, you have to implement a dictionary service. The service should provide two operations:

- search for a word; **returns** `true|false`
- search for all the words of a text file; **returns** a list of words that are not in the dictionary.

You can find an English dictionary here (81027 words).

You have to provide a flexible implementation that allows the service to use dictionaries based on various data structures such as an array list, balanced binary search tree, or hash table.

```
public static IDictionary createDictionary( DictionaryType dtype ){
   IDictionary dictionary = null;
   switch( dtype ){
      case ARRAY_LIST: dictionary = ArrayListDictionary.newInstance(); break;
      case HASH_SET  : dictionary = HashSetDictionary.newInstance(); break;
      case TREE_SET  : dictionary = TreeSetDictionary.newInstance(); break;
   }
   return dictionary;
}
```

In this lab, you are required to implement the dictionary using `ArrayList<String>` for storing the words and `Collections.binarySearch` for searching. This class should be a **Singleton** (only a single instance is permitted!)

A potential implementation is illustrated in the class diagram shown in Fig. 2.
Help:

- Explanation of the diagram and recommended implementation order:
  - `IDictionary` - interface
  - `ArrayListDictionary` - class; **Singleton**; implements the `IDictionary` interface
  - `DictionaryType` - enum
  - `DictionaryProvider` - utility class; role: creation of dictionary instances based on DictionaryType
  - `DictionaryService` - class; composition relationship with `IDictionary`
- `ArrayListDictionary` class should be tested before implementing the `DictionaryService`!

Examples for using the service:

1. Search for a word: `service.findWord(word)`

```
DictionaryService service =
                      new DictionaryService(DictionaryType.ARRAY_LIST);
Scanner scanner = new Scanner(System.in);
while( true ) {
   System.out.print("Word to find ( Enter <end> for exit): ");
   String word = scanner.nextLine();
   if( word.equalsIgnoreCase( "end")){
       break;
   }
   System.out.println(" Find(" + word + "): " + service.findWord(word));
}
scanner.close();
```

2. Search for all the words of a text file:
   `service.findWordsFile("text_to_find.txt")`

```
DictionaryService service =
                      new DictionaryService(DictionaryType.ARRAY_LIST);
System.out.println("Unknown words from a file: ");
System.out.println( service.findWordsFile( "text_to_find.txt"));
```

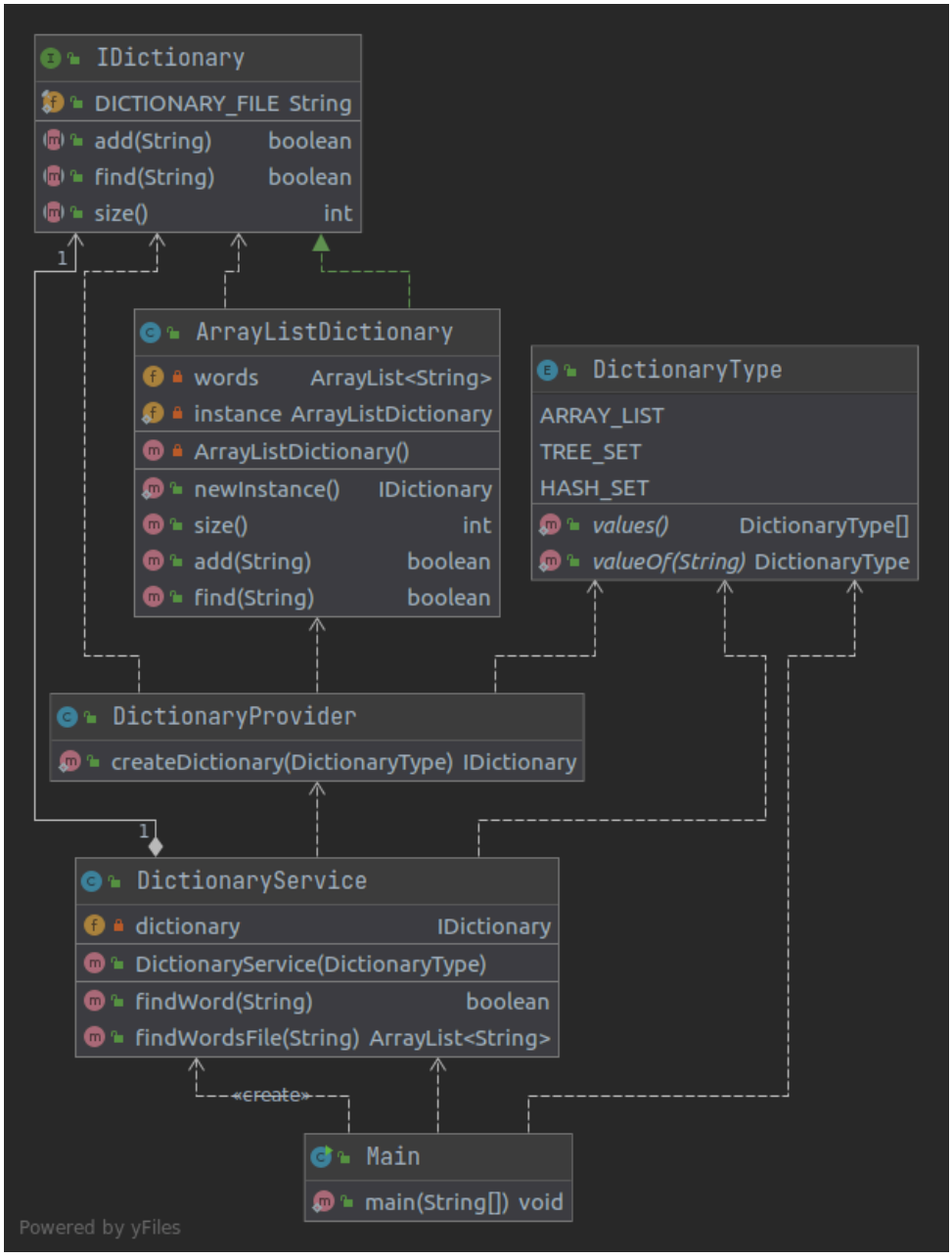

Fig.2. Dictionary service class diagram