UNIVERSIDAD DE MÁLAGA
Dept. of Languages and Computer Science

Advanced Programming II
Lab 2.2. Mutable and immutable queues

In this practice, you have to develop different implementations of a queue in Scala in order to understand different strategies for its representation and optimization:

1. **A mutable queue based on an array**: you'll use an `ArrayBuffer` to manage the queue structure efficiently.
2. **A simple immutable queue**: you'll implement a queue based on a structure of our own, where elements are extracted from one end and recursively added to the other.
3. **An efficient immutable queue**: you will implement an optimized version of a queue by using two lists, allowing direct extraction from the front and efficient insertion at the other end.

Exercise 1: Implementing Mutable Queues on Arrays

- Implement a `trait MutableQueue[T]` with the basic operations:
  - `enqueue(elem: T): Unit` that adds an element to the end of the queue.
  - `dequeue(): Option[T]` that pulls an item from the front of the queue, if it exists.
  - `isEmpty: Boolean` that indicates whether the queue is empty or not.
- Implement a `class ArrayQueue[T]` that extends `MutableQueue[T]`, using an `ArrayBuffer[T]` as the underlying structure.
- In addition to the trait methods, the class `ArrayQueue[T]` will provide a constructor without arguments and one that accepts multiple values (`T*`) as well as the `toString`, `equals`, and `hashCode` methods.

Exercise 2: Implementing Simple Immutable Queues

- Implement a `trait ImmutableQueue[T]` with the basic operations:
  - `enqueue(elem: T): ImmutableQueue[T]` that returns a new queue with the added element.
  - `dequeue(): (Option[T], ImmutableQueue[T])` that returns a tuple with the extracted element and the new queue.
  - `isEmpty: Boolean` that indicates whether the queue is empty.
- Implement a class `SimpleQueue[T]` that extends `ImmutableQueue[T]` and provides a constructor with no arguments and another with `T*` and `toString`, `equals`, and `hashCode` methods.

Exercise 3. Implementing Efficient Immutable Queues

- Implement a `class EfficientQueue[T]` that extends `ImmutableQueue[T]` and uses two lists (`front` and `rear`) to store queue items. If we visualize the queue as a sequence of elements and divide it at some point, in front we would have the elements at the front of the queue, in rear the elements in the queue. To make it more efficient, we will keep the elements to be reverted, so that:
  - To extract an element we extract it from the front head. When `front` is empty, before taking it out, we reverse `rear` and transfer it to `front`.
  - To insert an element we insert it into the rear head
- The class will provide a constructor without arguments and another with `T*`, and redefinitions of the `toString`, `equals`, and `hashCode` methods.