TP5 Java Avancé - Faites la queue

Max Ducoudré - INFO2

Exercice 2 - Fifo

1. On souhaite écrire une classe Fifo générique (avec une variable de type E) dans le package fr.uge.fifo prenant en paramètre une capacité (un entier), le nombre d'éléments maximal que peut stocker la structure de données.

On fabrique une classe Fifo avec un type paramétré :

```
public class Fifo<E> {
    private final static int DEFAULT_CAPACITY = 2;
    private final int capacity;
    private int size = 0;

    public Fifo(int capacity) {
        if(capacity < 1) throw new IllegalArgumentException("The capacity must be
> 0");
        this.capacity = capacity;
    }
    public Fifo() {
        this(DEFAULT_CAPACITY);
    }

    public void offer(E element) {
        Objects.requireNonNull(element);
        size++;
    }

    public int size() {
        return size;
    }
}
```

Cette classe vas contenir un champ `size` qui stockera la taille courant de la collection fifo (par défaut à 0). Cet entier est ensuite incrémenté à chaque ajout.

3. On souhaite écrire une méthode poll qui retire un élément du tableau circulaire. Que faire si la file est vide ?
   Quand la file est vide, elle retourne `null`. Voici une implémentation des méthodes `poll` et `offer` :

```
public class Fifo<E> {
    private final static int DEFAULT_CAPACITY = 16;
    private final int capacity;
    private int size = 0;
```

```
    private int tail = 0;
    private int head = 0;
    private Object[] elements;

    public Fifo(int capacity) {
        if(capacity < 1) throw new IllegalArgumentException("The capacity must be
> 0");
        this.capacity = capacity;
        elements = new Object[capacity];
    }

    /* [...] */


    public void offer(E element) {
        Objects.requireNonNull(element);
        if(size >= capacity) throw new IllegalStateException("The FIFO is full
!");

        elements[tail] = element;
        tail++;
        if(tail >= capacity) {
            tail = 0;
        }
        size++;
    }

    @SuppressWarnings("unchecked")
    public E poll() {
        if(size <= 0) return null;
        var element = elements[head];
        head++;
        if(head >= capacity) {
            head = 0;
        }
        size--;
        return (E) element;
    }

    @SuppressWarnings("unchecked")
    public E peek() {
        if(size <= 0) return null;
        return (E) elements[head];

    }
}
```

Ici, on utilise un tableau d'`Object` et deux index `tail` et `head` pour manipuler la liste FIFO.

4. Rappelez ce qu'est un memory leak en Java et assurez-vous que votre implantation n'a pas ce comportement indésirable.
   Un memory leak en Java arrive quand des éléments sont encore présents en mémoire mais plus du tout

utilisé. Ici, notre implémentation ne met pas à `null` les éléments qui ont étés retirés. Voici une nouvelle implémentation de la méthode `poll` :

```
public class Fifo<E> {
    /* [...] */
    @SuppressWarnings("unchecked")
    public E poll() {
        if(size <= 0) return null;
        var element = elements[head];
        elements[head] = null; // Set to null to avoid memory leak
        head++;
        if(head >= capacity) {
            head = 0;
        }
        size--;
        return (E) element;
    }
}
```

5. On souhaite agrandir le tableau circulaire dynamiquement en doublant sa taille quand le tableau est plein. Attention, il faut penser au cas où le début de la liste (head) a un indice qui est supérieur à l'indice indiquant la fin de la file (tail). Modifier votre implantation pour que le tableau s'agrandisse dynamiquement en ajoutant une méthode resize.

La méthode `resize` est appellée lorsque la taille du FIFO dépasse la moitié de la capacité. Cette méthode double la taille du tableau et copie les éléments dans le nouveau tableau. Tout en gardant le même ordre des éléments avec les index `tail` et `head`:

```
public class Fifo<E> {
    /* [...] */
   private void resize() {
        var newElements = new Object[capacity*2];

        if(head < tail) {
            // Si l'index head est inférieur à l'index head, on ajoute
            for(int i = 0; i < elements.length; i++) {
                newElements[i] = elements[i];
            }
        } else {
            // Sinon, on ajoute les éléments de head à la fin du tableau
            for(int i = head; i < elements.length; i++) {
                newElements[i-head] = elements[i];
            }
            // Et on ajoute les éléments du début du tableau à tail
            for(int i = 0; i < tail; i++) {
                newElements[capacity-head+i] = elements[i];
            }
        }

        elements = newElements;
```

```
            head = 0;
            tail = size;
            capacity *= 2;
        }

        public void offer(E element) {
            Objects.requireNonNull(element);
            //if(size >= capacity) throw new IllegalStateException("The FIFO is full
!");
            if(size*2 >= capacity) {
                resize();
            }

            elements[tail] = element;
            tail++;
            if(tail >= capacity) {
                tail = 0;
            }
            size++;
        }
    }
```

6. On souhaite ajouter une méthode d'affichage qui affiche les éléments dans l'ordre dans lequel ils seraient sortis en utilisant poll. L'ensemble des éléments devra être affiché entre crochets ('[' et ']') avec les éléments séparés par des virgules (suivies d'un espace).
   Voici une implémentation de la méthode toString pour afficher tous les éléments dans l'ordre où ils seraient sortis :

```
public class Fifo<E> {
    /* [...] */
    @Override
    public String toString() {
        if(size == 0) return "[]";

        var sb = new StringBuilder();
        sb.append("[");
        for(int i = head; i != tail; i++) {
            if(i == size) i = 0;
            sb.append(elements[i].toString());
            sb.append(", ");
        }

        sb.delete(sb.length()-2, sb.length());
        sb.append("]");
        return sb.toString();
    }


}
```

7. En fait, le code que vous avez écrit est peut-être faux (oui, les testent passent, et alors ?)... Le calcul sur les entiers n'est pas sûr/sécurisé en Java (ou en C, car Java à copié le modèle de calcul du C). En effet, une opération '+' sur deux nombres suffisamment grand devient négatif. Si celà peut se produire dans votre code, modifiez-le ! L'astuce est d'utiliser deux indices, pour qu'il soit correct.

Voici une implémentation de toString avec un stream :

```
public class Fifo<E> {
    /* [...] */
    @Override
    public String toString() {
        return Arrays.stream(elements)
            .filter(Objects::nonNull)
            .sorted(Comparator.comparingInt(o -> {
                    for(int i = head; i != tail; i++) {
                        if(i == size) i = 0;
                        if(elements[i].equals(o)) return 1;
                    }
                    return -1;
            }))
            .map(Object::toString)
            .collect(Collectors.joining(", ", "[", "]"));
    }
}
```

8. Rappelez quel est le principe d'un itérateur. Quel doit être le type de retour de la méthode iterator() ? Implanter la méthode iterator(). Vérifier que les tests marqués "Q8" passent. Note : ici, pour simplifier le problème, on considérera que l'itérateur ne peut pas supprimer des éléments pendant son parcours.

Un iterator permet d'itérer à travers une collection et a 2 méthodes : `hasNext` et `next`. Voici une implémentation de la méthode `iterator` :

```
public class Fifo<E> {
    /* [...] */
        public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int index = head;
            private int count = 0;

            @Override
            public boolean hasNext() {
                return count < size;
            }

            @SuppressWarnings("unchecked")
            @Override
            public E next() {
                if(!hasNext()) throw new NoSuchElementException();
                var element = elements[index];
                index++;
```

```
                if(index >= capacity) {
                    index = 0;
                }
                count++;
                return (E) element;
            }
        };

}
```

9. On souhaite que le tableau circulaire soit parcourable en utilisant une boucle for-each-in. Quelle interface doit implanter la classe Fifo ?

Il faut implanter l'interface `Iterable` pour pouvoir utiliser une boucle for-each-in. Pour cela il faut au moins que la méthode `iterator` soit implémentée.`

```
public class Fifo<E> implements Iterable<E> {
    /* [...] */
}
```

10. Enfin, il existe déjà en Java une interface pour les files d'éléments, java.util.Queue, on souhaite maintenant que notre implantation de tableau circulaire Fifo implante cette interface.

On modifie la classe `Fifo` pour la faire implémenter `Queue` et hériter de `AbstractQueue` et on modifie la valeur de retour de `offer` pour être cohérent avec l'interface Queue. (Ici, `offer` vas tout le temps renvoyer `true` car notre implémentation ne permet pas de refuser un élément) :

```
public class Fifo<E> extends AbstractQueue<E>  implements Iterable<E>, Queue<E>{
    /* [...] */
    public boolean offer(E element) {
        /* [...] */
        return true;
    }
}
```