Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

Introduction à l'informatique II (ITI 1521)
Examen final

Instructeur: Marcel Turcotte

Avril 2007, durée: 3 heures

Identification

Nom : _____ Prénom : _____

Numéro d'étudiant : _____ Signature : _____

Consignes

1. Livres fermés ;
2. Sans calculatrice ou toute autre forme d'aide ;
3. Commentez vos réponses afin d'obtenir des points partiels ;
4. Écrivez lisiblement, votre note en dépend ;
5. Répondez sur ce questionnaire, utilisez le verso des pages si nécessaire, mais vous ne pouvez remettre aucune page additionnelle ;
6. Ne retirez pas l'agrafe.

Barème

Question	Maximum	Résultat
1	20	
2	10	
3	15	
4	15	
5	10	
6	10	
7	10	
8	10	
Total	**100**	

Question 1 : Applications des files (20 points)

JStock est une application servant à la gestion d'un portfolio d'**actions** en bourse. La vente d'actions (parts égales du capital d'une entreprise) permet la collecte de fonds. L'ensemble des actions d'une entreprise s'appelle le stock. Le prix des actions varie constamment. Un actionnaire fait un **gain de capital** s'il vend ses actions à un prix plus élevé que le prix auquel il les a achetés ; ou encore, il subit une **perte de capital** si le prix de vente est inférieur au prix auquel il les a achetées.

Lors de la vente d'actions, le calcul du gain de capital est simple si toutes les actions ont été achetées au même prix (à la suite d'une seule transaction par exemple). Cependant, ce calcul sera plus complexe si les actions ont été acquises par le biais de plusieurs transactions. Dans ce cas, selon les principes de gestions standards, il faut vendre les actions les plus âgées d'abord.

Par exemple, un actionnaire achète 100 actions à 20 $ chacune lors d'une première transaction, puis il achète 20 actions à 24 $ chacune lors d'une seconde transaction, puis achète 200 actions à 36 $ chacune lors d'une troisième transaction, pour finalement vendre 150 actions à 30 $ l'action. Dans ce cas, voici le détail du calcul du gain de capital : $100 \times (30-20) + 20 \times (30-24) + 30 \times (30-36) = 940$ dollars.

L'application **JStock** facilite la gestion d'un portefeuille d'actions. Pour rendre le problème simple, **JStock** contient les actions d'un seul stock (d'une seule entreprise). Cependant, les actions sont généralement acquises par le biais de plusieurs transactions.

Toutes les actions qui ont été achetées dans une même transaction sont sauvegardées dans un objet **Transaction**. Le nombre d'actions et le prix de chacune sont spécifiés lors de la création d'un objet **Transaction**. La classe possède une méthode **getShares**, qui retourne le nombre d'actions, ainsi qu'une méthode **getSharePrice** qui retourne le prix individuel. Il y a aussi une méthode **sell**, qui réduit le nombre d'actions selon la valeur passée en paramètre.

Vous devez compléter l'implémentation de la classe **JStock** se trouvant aux pages 4 et 5. **JStock** utilise une file afin de sauvegarder les transactions d'un portefeuille d'actions.

A. Implémentez la méthode **void buy(int num, int sharePrice)**. Cette dernière ajoute une nouvelle transaction à l'arrière de la file. Les valeurs des paramètres sont utilisées pour la création de la nouvelle transaction ;

B. Implémentez la méthode **int sell(int num, int sharePrice)**. Cette dernière met à jour la file de transactions afin de réduire le nombre total d'actions de la valeur **num**. La méthode retourne le gain de capital (un gain négatif est interprété comme une perte) ;

C. Implémentez la méthode **int getValue()**, qui retourne la valeur totale du portefeuille. La valeur totale est la somme de la valeur des transactions. La valeur d'une transaction est simplement le produit du nombre d'actions par le prix de chacune.

Note : lorsque vous utilisez une file, vous n'avez accès qu'aux méthodes publiques.

Assumez l'existence d'une classe nommée **LinkedQueue** réalisant l'interface **Queue** ci-bas.

```
public interface Queue<E> {

    /**
     * Retourne true si cette file est vide.
     *
     * @return true si cette file est vide.
     */

    public abstract boolean isEmpty();

    /**
     * Retourne une référence vers l'élément avant; sans le retirer.
     *
     * @return La valeur de l'élément avant, sans retirer l'élément.
     */

    public abstract E peek() throws EmptyQueueException;

    /**
     * Ajoute un élément à l'arrière de la file.
     *
     * @throws FullQueueException si cette file est pleine.
     */

    public abstract void enqueue( E o ) throws QueueOverflowException;

    /**
     * Retire et retourne l'élément avant de cette file.
     *
     * @return l'élément avant de cette file.
     * @throws EmptyQueueException si cette file est vide.
     */

    public abstract E dequeue() throws EmptyQueueException;
}
```

Voici la classe **Transaction**.

```
public class Transaction {

    private int shares;
    private int sharePrice;

    public Transaction( int shares, int sharePrice ) {
        this.shares = shares;
        this.sharePrice = sharePrice;
    }
    public int getShares() {
        return shares;
    }
    public void sell( int num ) {
        if ( num < 0 || num > shares ) {
            throw new IllegalArgumentException( Integer.toString( num ) );
        }
        shares = shares - num;
    }
    public int getSharePrice() {
        return sharePrice;
    }
}
```

Voici la classe **JStock**.

```
public class JStock {

    private Queue<Transaction> myShares;

    public JStock() {
        myShares = new LinkedQueue<Transaction>();
    }

    public void buy( int num, int sharePrice ) {









    } // Fin de buy
```

```
    public int sell( int num, int sharePrice ) {

    } // Fin de sell

    public int getValue() {

    } // Fin de getValue
} // Fin de JStock
```

Question 2 : CircularQueue (10 points)

Reproduisez le résultat affiché à l'écran pour chacun des deux appels à la méthode **dump** des énoncés qui suivent. La déclaration de classe **CircularQueue** se trouve aux pages 7 et 8.

```
CircularQueue<Integer> q;
q = new CircularQueue<Integer>( 4 );

int i = 0;
while ( ! q.isFull() ) {
    i = i + 1;
    q.enqueue( new Integer( i ) );
}

if ( ! q.isEmpty() ) {
    q.dequeue();
}
if ( ! q.isEmpty() ) {
    q.dequeue();
}

while ( ! q.isFull() ) {
    i = i + 1;
    q.enqueue( new Integer( i ) );
}
q.dump();

while ( ! q.isEmpty() ) {
    q.dequeue();
}
q.dump();
```

Premier appel :	**Deuxième appel :**

```
public class CircularQueue<E> implements Queue<E> {

    public static final int DEFAULT_CAPACITY = 100;
    private final int MAX_QUEUE_SIZE;
    private E[] elems;
    private int front, rear;

    public CircularQueue() {
        this( DEFAULT_CAPACITY );
    }
    public CircularQueue( int capacity ) {
        if ( capacity < 0 ) {
            throw new IllegalArgumentException( Integer.toString( capacity ) );
        }
        MAX_QUEUE_SIZE = capacity;
        elems = (E []) new Object[ MAX_QUEUE_SIZE ];
        front = 0;
        rear = -1; // Représente une file vide
    }

    public boolean isEmpty() {
        return ( rear == -1 );
    }
    public boolean isFull() {
        return ( ! isEmpty() ) && nextIndex( rear ) == front;
    }
    private int nextIndex(int index) {
        return ( index+1 ) % MAX_QUEUE_SIZE;
    }

    public void dump() {

        System.out.println( "MAX_QUEUE_SIZE = " + MAX_QUEUE_SIZE );
        System.out.println( "front = " + front );
        System.out.println( "rear = " + rear );

        for ( int i=0; i<elems.length; i++ ) {
            System.out.print( "elems["+i+"] = " );
            if ( elems[ i ] == null ) {
                System.out.println( "null" );
            } else {
                System.out.println( elems[ i ] );
            }
        }

        System.out.println();
    }
```

```
    public void enqueue( E o ) {

        if ( o == null ) {
            throw new IllegalArgumentException( "null" );
        }

        if ( isFull() ) {
            throw new QueueOverflowException();
        }

        rear = nextIndex( rear );

        elems[ rear ] = o;
    }

    public E dequeue() {

        if ( isEmpty() ) {
            throw new EmptyQueueException();
        }

        E result = elems[ front ];
        elems[ front ] = null; // Pour la gestion de la mémoire

        if ( front == rear ) { // Suite à cet appel, la file sera vide
            front = 0;
            rear = -1;
        } else {
            front = nextIndex( front );
        }

        return result;
    }

} // Fin de CircularQueue
```

Question 3 : splitAfter (15 points)

Complétez l'implémentation de la méthode d'instance **LinkedList<E> splitAfter(E obj)**. La méthode **splitAfter** sectionne cette liste en deux parties. La liste originale contient alors les éléments du début de la liste jusqu'à et incluant l'occurrence la plus à gauche de l'élément **obj** alors que le reste est retourné dans une nouvelle liste. L'exception **IllegalArgumentException** est lancée si **obj** n'est pas trouvé.

L'implémentation de la classe **LinkedList** est la même que celle du devoir 5.

- Une liste débute toujours par un noeud factice (dummy node), qui marque le début de la liste. Le noeud factice ne contient aucune valeur. La liste vide ne contient qu'un noeud factice ;
- Pour cette question, les noeuds de la liste sont doublement chaînés ;
- Ici, la liste est circulaire, c'est-à-dire que la référence **next** du dernier noeud de la liste pointe vers le noeud factice, et que la référence **previous** du noeud factice pointe vers le dernier élément de la liste. Dans le cas de la liste vide, le noeud factice est le premier et le dernier élément, ainsi ses variables **previous** et **next** le désignent.
- Enfin, puisque le dernier noeud est facilement accessible, en effet, c'est le noeud qui précède le noeud factice, l'en-tête n'a pas de pointeur vers le noeud arrière (tail).

Écrivez votre réponse directement dans la classe **LinkedList** sur la page qui suit. Aucun appel de méthode n'est permis ; à l'exception des appels aux constructeurs.

Suggestion : dessinez les diagrammes de mémoire pour tous les cas possibles.

```
public class LinkedList<E> {
    private static class Node<E> { // Noeuds doublement chaînés
        private E value;
        private Node<E> previous;
        private Node<E> next;
        private Node( E value, Node<E> previous, Node<E> next ) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }
    private Node<E> head;
    private int size;
    public LinkedList() {
        head = new Node<E>( null, null, null );
        head.next = head.previous = head;
        size = 0;
    }

    public LinkedList<E> splitAfter( E obj ) {
```

```
    } // Fin de splitAfter
} // Fin de LinkedList
```

Question 4 : Iterator (15 points)

Les deux parties de cette question portent sur la classe **LinkedList** (pages 13–14) et son itérateur (page 15).

A. Complétez l'implémentation de la méthode **Iterator<E> copy()** de la classe interne **LinkedListIterator** sur la page qui suit. Cette méthode retourne une copie de l'itérateur ;

B. Dans la classe **Test** ci-dessous, complétez l'implémentation de la méthode **compress**. Celle-ci transforme la liste en entrée afin de ne conserver qu'une copie des éléments consécutifs qui sont égaux. Soit **l** une liste contenant les éléments suivants : "a,a,a,a,b,b,b,b,c,d,d,a". À la suite d'un appel à la méthode **compress**, le contenu de la liste sera "a,b,c,d,a".
- Vous **devez** utiliser des itérateurs pour traverser les listes ;
- La **seule** méthode de la classe **LinkedList** que vous pouvez utiliser est la méthode **Iterator<E> iterator()** ;
- Vous pouvez utiliser toutes les méthodes d'interface **Iterator**.

```
public class Test {

    public static <E> void compress( LinkedList<E> xs ) {









    } // Fin de compress
} // Fin de Test
```

```
import java.util.ConcurrentModificationException;

public class LinkedList<E> {

    private class LinkedListIterator implements Iterator<E> {

        private Node<E> current;
        private int expectedModCount;

        private LinkedListIterator() {
            expectedModCount = modCount;
            current = head;
        }

        public E next() { ... }
        public boolean hasNext() { ... }
        public void remove() { ... }

        public Iterator<E> copy() {









       }

        private void checkConcurrentModification() {
            if ( expectedModCount != modCount ) {
                throw new ConcurrentModificationException();
            }
        }
    } // Fin de LinkedListIterator
```

```
    private static class Node<E> { // Noeuds doublement chaînés
        private E value;
        private Node<E> previous;
        private Node<E> next;
        private Node( E value, Node<E> previous, Node<E> next ) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }

    private Node<E> head; // Pointe vers le noeud factice
    private int modCount; // Pour l'implémentation "fail-fast"

    // Constructeur

    public LinkedList() {
        head = new Node<E>( null, null, null ); // Noeud factice
        head.next = head.previous = head;
        modCount = 0;
    }

    // Retourne un itérateur pour cette liste

    public Iterator<E> iterator() { ... }

    // Les méthodes de la classe LinkedList seraient ici,
    // mais ne peuvent être utilisées.

} // Fin de LinkedList
```

```
public interface Iterator<E> {

    /**
     * Retourne true si cette itération a encore des éléments à retourner.
     * (Autrement dit, retourne true si un appel à next ne lancera pas
     * une exception.)
     *
     * @return true s'il y a encore des éléments à retourner.
     */

    public abstract boolean hasNext();

    /**
     * Retourne le prochain élément de l'itération.
     *
     * @return le prochain élément de l'itération
     * @exception NoSuchElementException s'il n'y a pas d'élément
     * à retourner.
     */

    public abstract E next();

    /**
     * Retire de la liste le dernier élément retourné par la méthode
     * next.  Il y aura au plus un appel à remove par appel à next.
     *
     * @exception IllegalStateException s'il n'y eu aucun appel à next,
     * ou que le nombre d'appels à remove excède le nombre d'appels à la méthode next.
     */

    public void remove();

    /**
     * Retourne une copie de cet itérateur.
     *
     * @return une copie de cet itérateur.
     */

    public Iterator<E> copy();

}
```

Question 5 : take (10 points)

Complétez l'implémentation de la méthode d'instance **take** de la classe **SinglyLinkedList** ci-dessous. La méthode **SinglyLinkedList<E> take(int n)** retourne une nouvelle liste contenant les **n** premiers éléments de cette liste. Cette instance demeure inchangée par un appel à la méthode **take**. Vous devez implémenter la méthode à l'aide de la technique présentée en classe pour l'implémentation de méthodes récursives à l'intérieur de la classe. Il y a donc une partie publique qui sert à lancer le premier appel à la méthode privée, récursive.

```
public class SinglyLinkedList<E> {
    private static class Node<E> {
        private E value;
        private Node<E> next;
        private Node( E value, Node<E> next ) {
            this.value = value;
            this.next  = next;
        }
    }
    private Node<E> first; // Variable d'instance
    public void addFirst( E item ) { ... }
    public void addLast( E item ) { ... }

    public SinglyLinkedList<E> take( int n ) {









    } // Fin de take

    private                      takeRec(                          ) {









    } // Fin de takeRec
} // Fin de SinglyLinkedList
```

Question 6 : findMax (10 points)

Complétez l'implémentation de la méthode **findMax**. Elle retourne la plus grande valeur d'une séquence (**Sequence**). Son implémentation est récursive. La classe **Sequence** implémente une liste chaînée dont les méthodes permettent l'écriture de méthodes récursives efficaces à l'extérieur de la classe. Voici les caractéristiques de cette classe.

- Les éléments d'une séquence (**Sequence**) réalisent l'interface **Comparable** ;
- La classe **Sequence** déclare les méthodes qui suivent.
  - **boolean isEmpty()** ; retourne **true** si et seulement si cette séquence est vide ;
  - **E head()** ; retourne une référence vers la valeur sauvegardée dans le premier noeud de la liste ;
  - **Sequence<E> split()** ; retourne le reste de cette séquence, l'instance ne contient alors qu'un élément ;
  - **void join(Sequence<E> other)** ; ajoute tous les éléments de la séquence désignée par **other** à la fin de cette séquence, conséquemment **other** est maintenant vide.

```
public class Q6 {
    public static < E extends Comparable<E> > E findMax( Sequence<E> xs ) {









    // Fin de findMax
// Fin de Q6
```

Question 7 : getPathLength (10 points)

Définissons la longueur du chemin menant à un noeud comme étant le nombre de liens qu'il faut traverser à partir de la racine pour se rendre à ce noeud. La longueur du chemin menant à la racine est 0. Vous devez implémenter la méthode **int getPathLength(E obj)** qui retourne la longueur du chemin menant au noeud où se trouve la valeur **obj** ou -1 si **obj** n'a pas été trouvé.

```
public class BinarySearchTree<E extends Comparable<E> > {
    private static class Node<E> {
        private E value;
        private Node<E> left;
        private Node<E> right;
        private Node( E value ) {
            this.value = value;
            left = null;
            right = null;
        }
    }
    private Node<E> root = null;

    // Réponse:
```

```
} // Fin de BinarySearchTree
```

Question 8 : Exceptions (10 points)

La méthode **readTransactions** lit un fichier contenant une transaction par ligne. Une transaction est constituée de deux entiers qui représentent le nombre d'actions et le prix de chaque action, respectivement. Un nombre positif d'actions représente un achat alors qu'un nombre négatif représente une vente.

A. Concevoir un nouveau type d'exception nommé **TransactionFileFormatException**. Il s'agit d'un type d'exception non vérifié. Il doit y avoir deux constructeurs : l'un sans paramètres et l'autre ayant un paramètre de type **String** afin de passer un message lorsqu'on lance une exception ;

B. Effectuez tous les changements nécessaires afin que la méthode **readTransactions** lance l'exception **TransactionFileFormatException** si l'entrée n'est pas valide. Suggestion : la classe **Scanner** possède une méthode, **boolean hasNextInt()**, qui retourne le booléen true si le prochain jeton de l'entrée est un entier ;

C. L'implémentation de la méthode **readTransactions** n'est pas valide parce la stratégie pour traiter les exceptions n'a pas été établie. Effectuez tous les changements nécessaires afin que la définition de la méthode soit valide (puisse être compilée sans erreurs). Effectuez ces changements directement dans le code source, sur la page qui suit. Voici les exceptions documentées.

- **FileInputStream(String name)** lance **FileNotFoundException** (une exception vérifiée) si le fichier n'existe pas, s'il s'agit d'un répertoire plutôt qu'un fichier régulier, ou pour toute autre raison qui empêche l'ouverture du fichier en mode lecture ;
- **String readLine()** lance **IOException** (une exception vérifiée) si une erreur en lecture survient ;
- **int nextInt()** lance **InputMismatchException** (une exception non vérifiée) si le prochain jeton n'est pas un entier, ou cet entier est trop grand pour le type **int** ;
- **int nextInt()** lance **NoSuchElementException** (une exception non vérifiée) si la fin de l'entrée a été atteinte ;
- **close** lance **IOException** (une exception vérifiée) à la suite d'une erreur d'entrées/sorties.

Réponse de la partie A.

Réponses aux parties B et C.

```
public class JStock {

    // ...

    public static JStock readTransactions( String fileName ) {


        FileInputStream fin;
        BufferedReader input;
        Scanner scanner;
        String line;
        JStock js;

        fin = new FileInputStream( fileName );

        input = new BufferedReader( new InputStreamReader( fin ) );

        js = new JStock();

        while ( ( line = input.readLine() ) != null ) {

            scanner = new Scanner( line ); // Analyseur lexical

            int a = scanner.nextInt(); // Retourne le prochain entier

            int b = scanner.nextInt(); // Retourne le prochain entier

            if ( a > 0 ) {
                js.buy( a, b );
            } else {
                js.sell( a, b );
            }

        }

        input.close();

        return js;

    } // Fin de readTransactions


} // Fin de JStock
```