Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

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

Instructeur: Marcel Turcotte

Avril 2012, durée: 3 heures

Identification

Nom, prénom : _____

Numéro d'étudiant : _____ Signature : _____

Consignes

1. **Lisez ces consignes ;**
2. Livres fermés ; Sans calculatrice ou toute autre forme d'aide ;
3. Les appareils électroniques, ou tout autre dispositif de communication, ne sont pas autorisés ; Tout appareil doit être éteint, rangé et hors de portée ; Quiconque contrevient au présent règlement peut être accusé de fraude scolaire ;
4. Répondez sur ce questionnaire, utilisez le verso des pages si nécessaire, mais vous ne pouvez remettre aucune page additionnelle ;
5. Écrivez lisiblement, votre note en dépend ; Commentez vos réponses ; Ne retirez pas l'agrafe.

Barème

Question	Maximum	Résultat
1	10	
2	10	
3	20	
4	20	
5	8	
6	13	
7	12	
8	7	
Total	**100**	

Question 1 : (10 points)

A. En Java, il y a deux façons de comparer des objets, par identité ou par contenu. Lequel des deux énoncés qui suivent permet de comparer des objets par identité ?

(a) **x == y**

(b) **x.equals(y)**

B. En Java, il y a deux paradigmes (façons) pour créer des structures de données dans lequelles ont peut sauvegarder des objets de plusieurs types, les classes paramétrées (« generics ») et l'héritage. Lequel des deux paradigmes suivants permet la détection des erreurs de types lors de la compilation ?

(a) Héritage

(b) Les classes paramétrées (« generics »)

C. L'énoncé "**cmp = eq(i, j)**" ci-dessous entraînera une erreur de compilation.
Vrai ou **faux**.

```
public class Test {
    public static boolean eq( int a, int b ) {
        boolean result;
        if ( a == b ) {
            result = true;
        } else {
            result = false;
        }
        return result;
    }
    public static void main( String[] args ) {
        long i, j;
        i = 5;
        j = 10;
        boolean cmp = eq( i, j );
    }
}
```

D. Plusieurs méthodes d'une même classe peuvent avoir le même nom, pour autant que les types des valeurs de retour sont différents.
Vrai ou **faux**

E. Le programme ci-dessous affiche **true**.
Vrai ou **faux**

```
public class Counter {
    private static int value = 0;
    public void incr() {
        value = value+1;
    }
    public int getValue() {
        return value;
    }
    public static void main( String[] args ) {
        Counter a, b;
        a = new Counter();
        a.incr();
        b = new Counter();
        b.incr();
        System.out.println( b.getValue() == 1 );
    }
}
```

F. Un constructeur n'a aucune valeur/type de retour.
Vrai ou **faux**

G. Une variable référence de type **T**, où **T** est une interface, peut désigner un objet de la classe **S** si **S**, ou l'une de ses superclasses, réalise l'interface **T**.
Vrai ou **faux**

H. Le nom **this** représente une référence à laquelle on a toujours accès dans une méthode d'instance et qui désigne l'objet lui-même.
Vrai ou **faux**

I. Une méthode qui pourrait lancer une exception de la classe **RuntimeException**, ou l'une de ses sous-classes, **doit** déclarer cette exception à l'aide d'une clause **throws**.
Vrai ou **faux**

J. Dans un arbre binaire de recherche, les valeurs dupliquées sont permises.
Vrai ou **faux**.

Question 2 : (10 points)

A. Quel terme désigne une méthode qui est exécutée automatiquement lors de la création d'un objet ?

(a) Setter
(b) Getter
(c) Initializer
(d) Constructor
(e) Mutator

Réponse :

B. Les _____ d'un objet définissent ses comportements.

(a) attributs
(b) espaces blancs
(c) variables
(d) méthodes
(e) noms

Réponse :

C. Pour une classe donnée, si une variable locale a le même nom qu'une variable d'instance.

(a) Il y aura une erreur de compilation
(b) Il y aura une erreur lors de l'exécution
(c) La variable locale masquera la variable d'instance
(d) La variable d'instance masquera la variable locale
(e) Les variables seront fusionnées

Réponse :

D. Quel mot clé indique qu'une classe est dérivée d'une autre par héritage ?

(a) super
(b) final
(c) extends
(d) inherits
(e) expands

Réponse :

E. Au sujet de l'implémentation **ArrayList** de l'interface **List**.

(a) Les insertions aux positions intermédiaires sont toujours rapides
(b) L'ajout d'un élément à la première position est toujours rapide
(c) Les accès en lecture aux positions intermédiaires sont toujours rapides
(d) Le retrait du premier élément est toujours rapide

Réponse :

Question 3 : (20 points)

A. En suivant les consignes présentées en classe, ainsi que dans vos notes de cours, dessinez les diagrammes de mémoire pour tous les objets, toutes les variables locales et le paramètre de la méthode **Property.main** immédiatement **avant** l'exécution de l'énoncé "aProperty = null".

```
public class Property {

    private String name;
    private double value;

    public Property( String name, double value ) {
        this.name = name;
        this.value = value;
    }

    public static void main( String[] args ) {

        Property aProperty;
        String aName;
        double aValue;

        aName = new String( "pi" );
        aValue = 3.14159265;

        aProperty = new Property( aName, aValue );

        aProperty = null;

    }
}
```

B. Identifiez cinq (5) erreurs ou avertissements (« warnings ») qui se produiront lors de la compilation du programme Java ci-dessous.

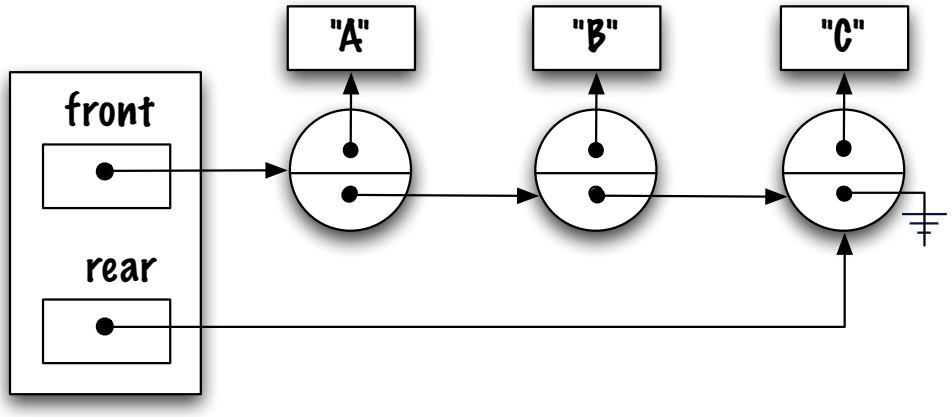
```
public class LinkedStack<E> {

    private static class Node<T> {

        private E value;
        private Node<T> next;

        private Node( T item, Node next ) {
            value = item;
            this.next = next;
        }

    }

    private Node<E> head;

    head == null;

    public static void main( String[] args ) {

        Node<E> p;
        p = head;

        while (p != null ) {
            System.out.println( p.value );
            p.next();
        }

    }

}
```

C. Étant donné l'implémentation partielle de la classe **LinkedQueue**.

```
public class LinkedQueue {

    private static class Node {
        private String value;
        private Node next;
        private Node( String value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Node front, rear;

    // ...
}
```

Modifiez le diagramme de mémoire ci-dessous afin de représenter le contenu de la mémoire suite à l'exécution de l'énoncé qui suit :

```
rear.next = new Node( "D", rear );
```



D. Étudiez le programme Java ci-dessous et donnez la sortie du programme.

```
1  public class Test {
2      public static void displayRatio( int a, int b ) {
3          if ( b == 0 ) {
4              throw new IllegalArgumentException( "zero" );
5          }
6          try {
7              System.out.println( "displayRatio: ratio is " + (a/b) );
8          } catch( IllegalArgumentException e1 ) {
9              System.out.println( "displayRatio: caught IllegalArgumentException" );
10         } catch( ArithmeticException e2 ) {
11             System.out.println( "displayRatio: caught ArithmeticException" );
12         }
13     }
14     public static void main( String[] args ) {
15         try {
16             displayRatio( 5, 0 );
17         } catch (RuntimeException e) {
18             System.out.println( "main: caught RuntimeException: " + e );
19         }
20     }
21 }
```

(a) Affiche : “main : caught RuntimeException : java.lang.IllegalArgumentException : zero”

(b) Termine abruptement et affiche la pile d’exécution qui suit :

```
Exception in thread "main" java.lang.IllegalArgumentException: zero
        at Test.displayRatio(Test.java:4)
        at Test.main(Test.java:16)
```

(c) Termine abruptement et affiche la pile d’exécution qui suit :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.displayRatio(Test.java:7)
        at Test.main(Test.java:16)
```

(d) Affiche “displayRatio : ratio is (5/0)”

(e) Affiche “displayRatio : caught IllegalArgumentException”

(f) Affiche “displayRatio : caught ArithmeticException”

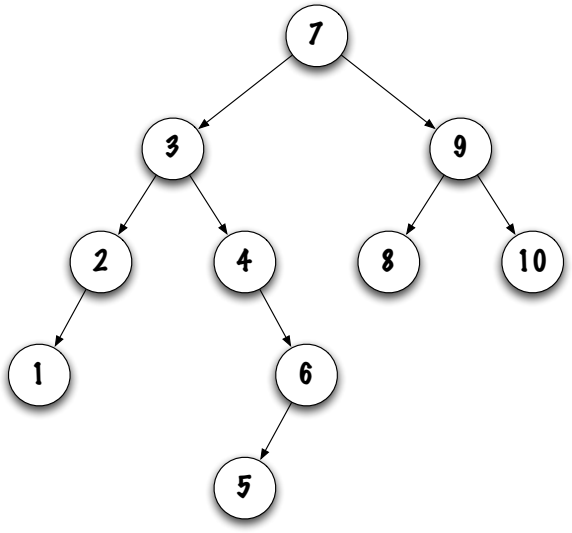
E. Complétez l'implémentation de la méthode d'instance récursive **sum** à l'aide de la technique « head+tail » vue en classe. La méthode **sum** retourne la somme des éléments de la liste. En particulier, l'exécution de la méthode principale affichera la valeur 10.

```
public class LinkedList {

    private static class Node {
        private int value;
        private Node next;
        private Node( int value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Node head;

    public void addFirst( int elem ) {
        head = new Node( elem, head );
    }

    public int sum() {

        int result = ____________________________________;

        return result;
    }

    private ________________ sum( ____________________________________ ) {

        ________________ result;

        if ( ____________________________________ ) { // Cas de base

            result = ____________________________________;

        } else { // General case

            ________________ s = ____________________________________; // Appel récursif

            result = ____________________________________;
        }
        return result;
    }

    public static void main( String[] args ) {
        LinkedList l;
        l = new LinkedList();
        for (int i=0; i<5; i++) {
            l.addFirst(i);
        }
        System.out.println( l.sum() );
    }

}
```

F. Soit la méthode **add**, vue en classe et reproduite ci-dessous, pour un arbre binaire de recherche.

```
private boolean add( Integer obj, Node current ) {

    boolean result;
    int test = obj.compareTo( current.value );

    if ( test == 0 ) {
        result = false; // already exists, not added
    } else if ( test < 0 ) {
        if ( current.left == null ) {
            current.left = new Node( obj );
            result = true;
        } else {
            result = add( obj, current.left );
        }
    } else {
        if ( current.right == null ) {
            current.right = new Node( obj );
            result = true;
        } else {
            result = add( obj, current.right );
        }
    }
    return result;
}
```

Donnez l'ordre dans lequel les valeurs 1, 2, 3, 4, 5, 6, 7, 8, 9 et 10 doivent être insérées dans un arbre binaire de recherche afin de produire l'arbre ci-bas. Les valeurs sont insérées dans un arbre initialement vide à l'aide d'appels répétés à la méthode **add** ci-dessus. Inscrivez votre réponse dans la boîte.



Question 4 : (20 points)

Un jeu de cartes comprend 52 cartes. Chaque carte possède une couleur (**suit**) et un rang (**rank**). Il y a quatre couleurs : 1 (pique, *spades*), 2 (coeur, *heart*), 3 (carreau, *diamond*) et 4 (trèfle, *club*), ainsi que 13 rangs : 1 (as, *ace*), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (valet, *jack*), 12 (reine, *queen*) et 13 (roi, *king*). Les tableaux ci-dessous donnent les valeurs entières utilisées afin de représenter les couleurs et les rangs des cartes pour cette question.

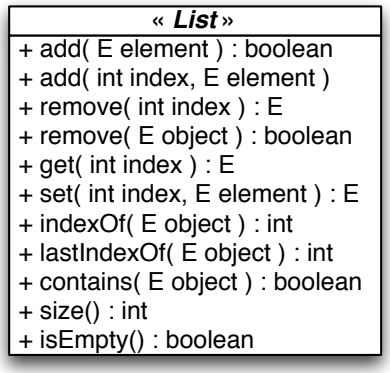
```
int[] suits = { 1, 2, 3, 4 };
int[] ranks = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 };
```

Donnez l'implémentation des classes **Card**, **Deck** et **Hand** étant donné les instructions qui suivent.

A. La classe **Card** possède des attributs afin mémoriser le rang et la couleur d'une carte. La classe **Card** réalise l'interface **Comparable<Card>**, vous devez donc implémenter la méthode **int compareTo(Card other)**. Cette dernière retourne la valeur -1, 0 ou 1 selon que cet objet est plus petit, égal, ou plus grand que l'objet désigné par **other**. Assurez que votre implémentation possède au moins un constructeur, ainsi que les méthodes d'accès (getter, setter) appropriées.

B. La classe **Deck** utilise un objet de la classe **ArrayList** afin de sauvegarder les 52 cartes d'un jeu. La classe **ArrayList** réalise l'interface **List**. La classe **Deck** possède un constructeur d'arité 0 (sans paramètre) qui doit créer et sauvegarder les 52 cartes du jeu. Les cartes sont créées dans l'ordre pique, coeur, carreau, puis trèfle. Vous devez implémenter les méthodes suivantes.

- **shuffle() :** doit battre (mélanger) les cartes du jeu. Chaque carte du jeu est remplacée par une carte choisie au hasard.
  Suggestion : utilisez la méthode **random** de la classe **java.lang.Math**, cette dernière retourne un nombre aléatoire dans l'intervalle [0,1), ou encore utilisez le générateur de nombre aléatoire **Random** de **java.util**, ce dernier possède une méthode **nextInt(int n)** qui retourne un entier choisi aléatoirement dans l'intervalle 0 (inclut) à **n** (exclut).
- **public Card dealCard()** : retire la dernière carte du jeu et la retourne. La méthode lance l'exception **IllegalStateException** si le jeu ne contient aucune carte au moment de l'appel.



C. La classe **Hand** utilise un objet de la classe **ArrayList** afin de sauvegarder des cartes (objets de la classe **Card**). Vous devez implémenter la méthode **int total()** qui retourne le nombre de cartes que possède cette main.

Question 5 : (8 points)

Vous devez concevoir une méthode de classe retournant la valeur **true** si **q1** et **q2** désignent des files possédant le même nombre d'objets, ces objets ont un contenu équivalent, et ils sont sauvegardés dans le même ordre.

```
public interface Queue<E> {
    public abstract boolean isEmpty();
    public abstract void enqueue( E value );
    public abstract E dequeue();
}
```

Pour cette question, il y a une interface nommée **Queue** et une implémentation, la classe **MysteryQueue**. Le constructeur **MysteryQueue()** initialise une nouvelle file vide. Cette implémentation peut contenir un nombre arbitraire d'éléments. Ne faites aucune autre hypothèse au sujet de son implémentation. En particulier, vous ne savez pas s'il s'agit d'une implémentation à l'aide d'éléments chaînés ou d'un tableau.

Pour la classe **Utils** ci-dessous, vous devez concevoir une méthode de classe qui retourne la valeur **true** si **q1** et **q2** désignent des files possédant le même nombre d'objets, ces objets ont un contenu équivalent au sens de la méthode **equals**, et ils sont sauvegardés dans le même ordre. La méthode retourne la valeur **false** sinon. Les objets désignés par **q1** et **q2** doivent demeurer inchangés suite à un appel à la méthode **eq**.

```
public class Utils {
    public static <E> boolean eq(Queue<E> q1, Queue<E> q2) {









    } // Fin de eq
} // Fin de Utils
```

Question 6 : (13 points)

Implémentez la méthode **remove(int from, int to)** pour la classe **LinkedList**. Cette méthode d'instance retire de cette liste tous les éléments situés dans l'intervalle de positions spécifiées et retourne ces éléments dans une nouvelle liste, dans l'ordre original. Voici les caractéristiques de la classe **LinkedList**.

- L'instance débute toujours par un noeud factice. Ce dernier marque le début de la liste. On n'y sauvegarde jamais une valeur. Une liste vide ne comprend que le noeud factice ;
- Les noeuds de la liste sont doublement chaînés ;
- La liste est circulaire, c'est-à-dire que la référence **next** du dernier noeud désigne le noeud factice, la référence **previous** du noeud factice désigne le dernier noeud de la liste. Si la liste est vide, le noeud factice sera à la fois le premier et le dernier noeud de la liste, ses références **previous** et **next** pointeront vers ce noeud unique ;
- Puisqu'on accède facilement au dernier élément de la liste, en effet, c'est le noeud qui précède le noeud factice, l'en-tête de la liste ne possède pas de pointeur arrière.

Exemple : si **xs** désigne une liste contenant les éléments suivants **[a,b,c,d,e,f]**, suite à l'appel de méthode **ys = xs.remove(2,3)**, la liste désignée par **xs** contient maintenant les éléments suivants **[a,b,e,f]**, et **ys** désigne une liste contenant ces éléments **[c,d]**.

Répondez sur la page qui suit. **Vous ne devez pas utiliser les méthodes de la classe LinkedList pour répondre à cette question. En particulier, vous ne devez pas utiliser les méthodes add() et remove().**

Suggestion : dessinez les diagrammes de mémoire associés.

```
public class LinkedList<E> {
    private static class Node<T> { // les noeuds de la liste
        private T value;
        private Node<T> previous;
        private Node<T> next;
        private Node( T value, Node<T> previous, Node<T> 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> remove( int from, int to ) {









    } // Fin de remove
} // Fin de LinkedList
```

Question 7 : (12 points)

Cette question porte sur le concept de file circulaire et le concept d'itérateur. La classe **CircularQueue** utilise un tableau circulaire de taille fixe afin de sauvegarder les éléments de la file. De plus, cette classe possède un itérateur.

```
public interface Iterator<E> {
    // Returns the next element in the iteration.
    public abstract E next();

    // Returns true if the iteration has more elements.
    public abstract boolean hasNext();
}
```

- Cette implémentation utilise un tableau circulaire de taille fixe ;
- Un objet de la classe **CircularQueue** possède une méthode **iterator** qui retourne un objet **CircularQueueIterator** qui réalise l'interface **Iterator** ;
- Un appel à la méthode **hasNext** d'un itérateur retourne **true** s'il y a encore des éléments à retourner dans cette itération, et **false** sinon.
- Un appel à la méthode **next** d'un itérateur retourne le prochain objet de l'itération. Spécifiquement, le premier appel retourne l'élément avant, le second appel retournera l'élément suivant, etc. Éventuellement, un appel à **next** retournera le dernier élément. Maintenant, un appel à **hasNext** retournera **false** et un appel à **next** entraînera une exception de type **NoSuchElementException**.

Complétez l'implémentation de la classe **CircularQueue**, ci-dessous et sur la page qui suit.

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

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

    public CircularQueue( int capacity ) {
            elems = (E[]) new Object[ capacity ];
            front = 0;
            rear = -1;
            size = 0;
    }

    // se poursuit sur la page suivante
```

```
    public boolean isEmpty() {
            return ( size == 0 );
    }

    public void enqueue( E value ) {
            rear = ( rear+1 ) % elems.length;
            elems[ rear ] = value;
            size++;
    }

    public E dequeue() {
        E savedValue = elems[ front ];
        elems[ front ] = null;
        size--;
        front = ( front+1 ) % elems.length;
        return savedValue;
    }

    private ___________ CircularQueueIterator implements Iterator<E> {

        private ___________ current = ___________;

        public E next() {

            if ( ____________________________________ ) {
                throw new NoSuchElementException();
            }









            return ________________________;
        }
        public boolean hasNext() {
            boolean result;




            return result;
        }
    } // Fin de CircularQueueIterator

    public ________________ iterator() {

        return ____________________________________;
    }

} // Fin de CircularQueue
```

Question 8 : (7 points)

Pour la classe **BinarySearchTree** ci-dessous, vous devez implémenter la méthode d'instance **public int getHeight(E elem)** qui retourne le nombre de liens qu'il faut suivre à partir de la racine de l'arbre pour arriver au noeud contenant la valeur **elem**, ou -1 si **elem** est absent de l'arbre.

```
public class BinarySearchTree< E extends Comparable<E> > {

    private static class Node<F> {

        private F value;
        private Node<F> left;
        private Node<F> right;

        private Node( F value ) {
            this.value = value;
            left = null;
            right = null;
        }
    }

    private Node<E> root = null;









}
```

(page blanche)