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 2011, durée: 3 heures

Identification

Nom, prénom : _____

Numéro d'étudiant : _____ Signature : _____

Consignes

1. **Lisez attentivement les consignes ;**
2. Examen à livres fermés ;
3. Sans calculatrice ou toute autre forme d'aide ;
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 ;
6. Commentez vos réponses ;
7. 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. L'énoncé « **cmp = compare(i, j)** » ci-dessous produira une erreur lors de la compilation.
Vrai ou **faux**.

```
public class Test {
    public static int compare( long a, long b ) {
        return a < b ? -1 : ( a == b ) ? 0 : 1;
    }
    public static void main( String[] args ) {
        int i, j, cmp;
        i = 5;
        j = 10;
        cmp = compare( i, j );
    }
}
```

B. Si **p** est de type **int**, le test de l'énoncé **if** ci-dessous produira une erreur lors de la compilation.
Vrai ou **faux**.

```
if ( p == null ) {
    System.out.println( "is empty" );
}
```

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

D. Chaque instance d'une classe possède ses propres variables d'instance.
Vrai ou **faux**

E. Lorsque l'on ajoute un constructeur à une classe, le constructeur sans argument qui est normalement fourni automatiquement n'est plus présent.
Vrai ou **faux**

F. Une variable référence de type **T** peut désigner un objet de la classe **T** ou l'une de ses superclasses.
Vrai ou **faux**

G. Java n'exige pas que l'on traite les exceptions si elles sont des sous-classes de **RuntimeException**.
Vrai ou **faux**

H. La déclaration **throws** lance une exception.
Vrai ou **faux**

I. Pour l'implémentation d'une liste simplement chaînée, la référence **tail** facilite l'implémentation de la méthode **addLast**.
Vrai ou **faux**.

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

Question 2 : (10 points)

A. Le nom d'une variable référence qui est toujours présente dans une méthode d'instance et qui désigne l'objet lui-même.

(a) self

(b) this

(c) object

(d) instance

(e) me

B. Lorsqu'une méthode d'une sous-classe possède la même signature qu'une méthode de la super-classe, c'est

(a) la surcharge de nom (overloading)

(b) la redéfinition d'une méthode (overriding)

(c) le chaînage de méthode (chaining)

(d) une erreur

C. Toutes les classes héritent directement ou indirectement de cette classe.

(a) Object

(b) Class

(c) Instance

(d) Root

(e) Super

D. Afin de retirer le premier noeud d'une liste simplement chaînée non vide, sans noeud factice,

(a) modifier la référence **next** du premier noeud et la faire pointer vers le noeud qui suit :

```
head.next = head.next.next;
```

(b) utiliser une variable référence temporaire que l'on fera pointer sur le noeud qui précède celui à retirer, changer le successeur de ce noeud afin qu'il désigne le noeud qui suit le premier noeud ;

(c) modifier la variable **head** afin qu'elle désigne le second noeud :

```
head = head.next;
```

(d) simplement retirer le premier noeud en affectant la valeur **null** à la variable **head** :

```
head = null;
```

E. Pour l'implémentation d'une file à l'aide d'éléments simplement chaînés,

(a) **front** désigne le premier élément et **rear** désigne le dernier élément ;

(b) **rear** désigne le premier élément et **front** désigne le dernier élément ;

(c) Ça n'a aucune importance, (a) et (b) sont deux implémentations aussi efficaces l'une que l'autre ;

(d) Aucune de ces réponses.

Question 3 : (20 points)

A. En respectant les consignes vues en classe, ainsi que dans les notes de cours, dessinez les **diagrammes de mémoire** pour tous les objets et toutes les variables locales de la méthode **ArrayList.init** suite à l'exécution de l'énoncé « **ys = xs** ».

```
public class ArrayList<E> {

    private E[] elems;
    private int size;

    public ArrayList( E value, int range, int capacity ) {
        elems = (E[]) new Object[ capacity ];

        for ( int i=0; i<range; i++ ) {
            elems[ i ] = value;
        }

        size = range;
    }

    public static void init() {
        String s;
        s = new String( "Quentin" );

        int r;
        r = 2;

        ArrayList<String> xs, ys;
        xs = new ArrayList<String>( s, r, 5 );

        ys = xs;
    }

}
```

B. Trouvez cinq (5) erreurs de compilation dans ce programme Java.

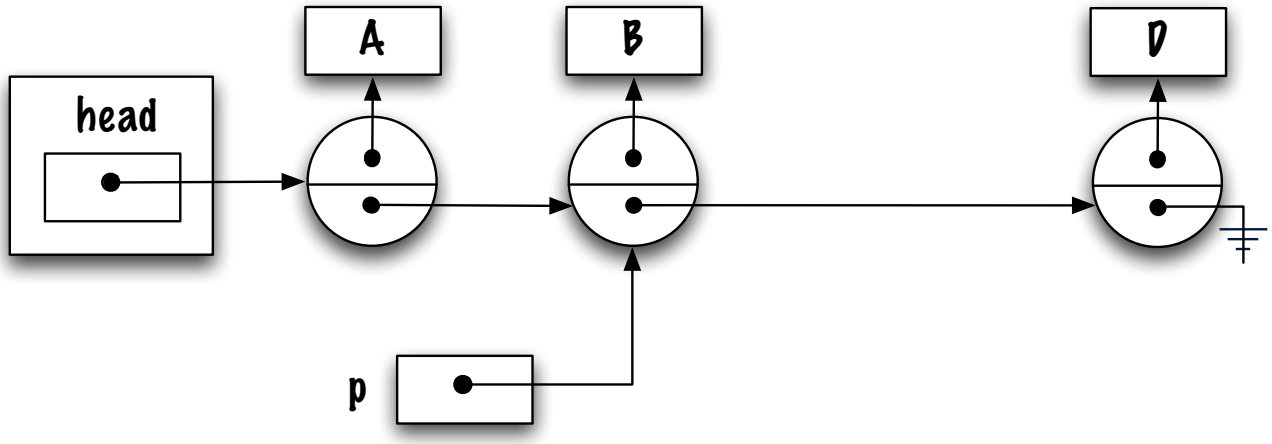
```
public class LinkedList {

    private static class Node<E> {

        private E value;
        private Node<E> next;

        private void Node( E value, Node<E> next ) {
            this.value = value;
            this.next = next;
        }

    }

    private Node<E> head;

    public static void main( String[] args ) {

        private Node<E> p;

        p = head;

        while ( p != 0 ) {
             System.out.println p.value;
             p++;
        }

    }

}
```

C. Étant donné cette implémentation partielle de la classe **LinkedList**.

```
public class LinkedList<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> head;

    // ...
}
```

Modifiez le diagramme de mémoire ci-dessous afin de représenter le contenu de la mémoire suite à l'exécution de l'énoncé qui suit [1] :

```
p = new Node<E>( C, p.next );
```



1. Assumez que **C** est une variable référence qui désigne un objet.

D. Analysez attentivement ce programme Java et donnez le résultat de son exécution.

```
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( "ratio is " + (a/b) );
8          } catch( IllegalArgumentException e1 ) {
9              System.out.println( "caught IllegalArgumentException" );
10         } catch( ArithmeticException e2 ) {
11             System.out.println( "caught ArithmeticException" );
12         }
13     }
14     public static void main( String[] args ) {
15         displayRatio( 5, 0 );
16     }
17 }
```

(a) Ce programme termine abruptement et affiche le contenu de 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:15)
```

(b) Ce programme termine abruptement et affiche le contenu de 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:15)
```

(c) Affiche « ratio is (5/0) »

(d) Affiche « caught IllegalArgumentException »

(e) Affiche « caught ArithmeticException »

E. Dessinez l'arbre binaire de recherche qui sera produit par l'ajout des éléments suivants, dans cet ordre, si l'on utilise la méthode **add** présentée en classe : 8, 12, 11, 4, 5, 1.

F. Attention, le programme Java qui suit comporte une erreur de logique ! Donnez le résultat qui sera affiché lors de l'exécution du programme.

```
public class LinkedList<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> head;
    public void addFirst( E elem ) {
        head = new Node<E>( elem, head );
    }
    public int size() {
        return size( head );
    }
    private int size( Node<E> current ) {
        int length = 0;
        if ( current == null ) {
            length = 0;
        } else {
            size( current.next );
            length++;
        }
        return length;
    }
    public static void main( String[] args ) {
        LinkedList<Integer> l;
        l = new LinkedList<Integer>();
        for ( int i=0; i<5; i++ ) {
            l.addFirst( i );
        }
        System.out.println( "The size of l is " + l.size() );
    }
}
```

(a) « The size of l is 0 »

(b) « The size of l is 1 »

(c) « The size of l is 4 »

(d) « The size of l is 6 »

(e) Des appels récursifs infinis vont causer un débordement de pile

Question 4 : (20 points)

Vous devez concevoir en Java l'implémentation des classes **Person**, **Customer**, et **PreferredCustomer** en suivant bien toutes instructions.

A. La classe **Person** possède des champs afin de sauvegarder le nom de la personne, son adresse et son numéro de téléphone. Assurez-vous d'y inclure au moins un constructeur, ainsi que les méthodes d'accès (getter et setter) appropriées.

B. Un client (**Customer**) est une personne (**Person**) qui possède un numéro de client, un total de la valeur de ses achats, ainsi qu'un champ indiquant si le client accepte sa participation à la liste de courriel (mailing list). Assurez-vous d'y inclure au moins un constructeur, ainsi que les méthodes d'accès appropriées.

C. Un client privilégié (**PreferredCustomer**) est un client (**Customer**) qui obtient un rabais selon le total de ses achats. Spécifiquement, lorsque le client privilégié a dépensé 500 $, il ou elle obtient un rabais de 5 % sur ses prochains achats, lorsque le client privilégié a dépensé 1,000 $, il ou elle obtient un rabais de 7.5 % sur ses prochains achats, finalement, lorsque le client privilégié a dépensé 2,000 $, il ou elle obtient un rabais de 10 % sur ses prochains achats. Ces objets possèdent une méthode **double getDiscountLevel()** qui retourne le pourcentage de rabais en fonction du total des achats. Assurez-vous d'inclure au moins un constructeur.

Question 5 : (8 points)

Implémentez la méthode de classe **static <E> void swap(Stack<E> xs, Stack<E> ys)**. Cette méthode échange le contenu de deux piles, **xs** et **ys**.

- Cette méthode doit fonctionner pour toute implémentation valide de l'interface **Stack** ;
- Vous pouvez assumer que les classes **DynamicArrayStack** et **LinkedStack** existent.

```
Stack<String> a, b;

a = new LinkedStack<String>();
a.push( "alpha" ); a.push( "beta" ); a.push( "gamma" );

b = new DynamicArrayStack<String>();
b.push( "blue" ); b.push( "green" ); b.push( "yellow" ); b.push( "black" );

System.out.println( a );
System.out.println( b );
swap( a, b );
System.out.println( a );
System.out.println( b );
```

L'exécution des énoncés ci-dessus devrait afficher ce qui suit :

```
[gamma,beta,alpha]
[black,yellow,green,blue]
[black,yellow,green,blue]
[gamma,beta,alpha]
```

```
public static <E> void swap( Stack<E> xs, Stack<E> ys ) {









}
```

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> {
        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)

Complétez l'implémentation de la classe **CircularStack**. Lisez attentivement toutes les directives, cette pile utilise un tableau circulaire !

```
public interface Stack<E> {

    // Adds an element onto the top of this stack
    public abstract void push( E element );

    // Removes and returns the top element of the stack
    public abstract E pop() throws java.util.EmptyStackException;

    // Returns true if and only if this stack is empty
    public abstract boolean isEmpty();
}
```

- Cette implémentation utilise un **tableau circulaire de taille fixe** ;
- Lorsque la pile est pleine, la méthode **push** remplace l'élément du dessous par le nouvel élément ajouté ;
- Ainsi, la méthode **push** ajoute toujours les éléments à la pile, même lorsqu'elle est pleine, mais les éléments les plus anciens (ceux du dessous) sont perdus ;
- Si n est la capacité de cette pile, alors la pile mémorise un maximum de n éléments (les derniers ajoutés) ;
- Le constructeur de la classe n'a qu'un paramètre, c'est la taille physique du tableau ;
- La valeur **null** est un élément valide.

```
public class CircularStack<E> implements Stack<E> {

    private E[] elems;
    private int top, size;

    public CircularStack( int capacity ) {


        elems = ____________________________________________;

        top = -1;
        size = 0;

    }

    public boolean isEmpty() {


        return ____________________________________;

    }
```

Se poursuit à la page qui suit.

Suite de la Question 7:

```
    public void push( E elem ) {

        top = ____________________________________________;

        elems[ top ] = elem;

        if ( ____________________________________ ) {

            size++;

        }
    }

    public E pop() {

        if ( isEmpty() ) {

            ____________________________________________________________;
        }

        E saved = elems[ top ];

        elems[ top ] = ____________________________________;

        if ( ____________________________________ ) {

            top = ____________________________________;

            size = 0;

        } else {

            top--;
            size--;

            if ( ____________________________________ ) {

                ____________________________________;

            }
        }

        return saved;
    }
} // Fin de CircularStack
```

Question 8 : (7 points)

Complétez l'implémentation de la méthode **isValid**. Cette méthode **récursive** d'instance retourne **true** si et seulement si tous les noeuds de l'arbre sont localement valides, et **false** sinon.

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

    private static class Node<F extends Comparable<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;

    public boolean isValid() {


        return isValid( ____________________ );

    }

    private boolean isValid( Node<E> current ) {
        boolean isValid = true;

        if ( current != null ) {

            if ( current.left != null ) {


                isValid = ____________________________________ && ____________________ ;

            }

            if ( ____________________________ ) {


                isValid = ____________________________________ && ____________________ ;

            }

        }

        return isValid;
    }
}
```

(page blanche)