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 to Computing II (ITI 1121)
MIDTERM EXAMINATION

Instructors: Nour El-Kadri, Guy-Vincent Jourdan, and Marcel Turcotte

February 2016, duration: 2 hours

Identification

Last name: ______________________ First name: ______________________

Student #: ______________ Seat #: ct_____ Signature: ______________________ Section: A or B or C

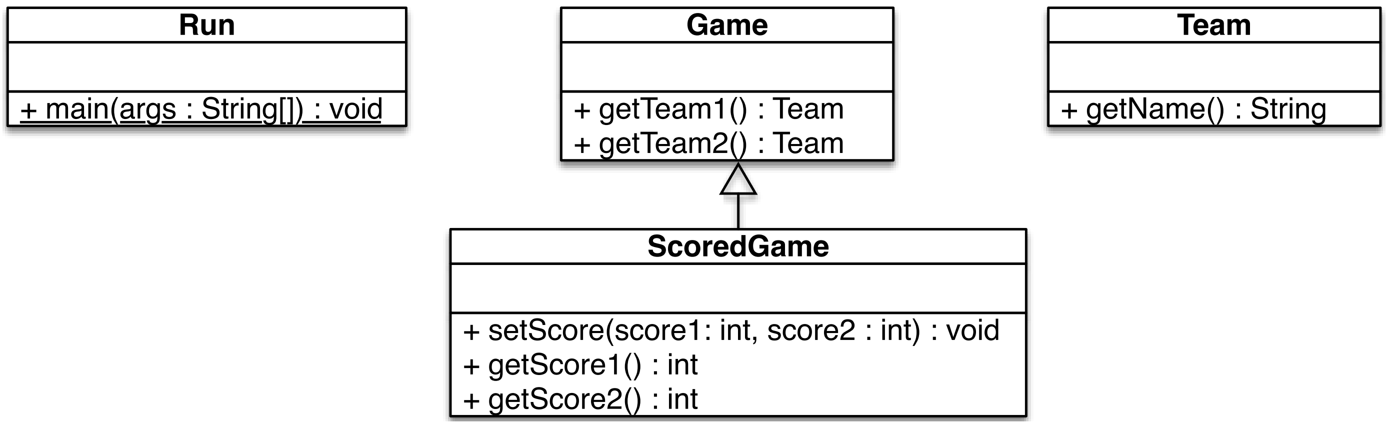
Instructions

1. This is a closed book examination.
2. No calculators, electronic devices or other aids are permitted.
   (a) Any electronic device or tool must be shut off, stored and out of reach.
   (b) Anyone who fails to comply with these regulations may be charged with academic fraud.
3. Write your answers in the space provided.
   (a) Use the back of pages if necessary.
   (b) You may not hand in additional pages.
4. Do not remove pages or the staple holding the examination pages together.
5. Write comments and assumptions to get partial marks.
6. Beware, poor hand writing can affect grades.
7. Wait for the start of the examination.

Marking scheme

Question	Maximum	Result
1	10	
2	15	
3	20	
4	20	
Total	**65**	

Question 1 (10 marks)



For this question, you must implement the classes **Game**, **ScoredGame**, and **Team**, according to the UML diagram above, and the instructions below. You must also make sure that your implementation is consistant with the provided test class **Run**.

A. An object of the class **Team** stores the name of a sport's team. Make sure to include a getter for this attribute.

B. The class **Game** describes the characteristics that are common to all games. Namely, a game has two teams (objects of the class **Team**). Make sure to include getters for these two attributes.

C. Finally, the class **ScoredGame** is a specialization of the class **Game**. An object of the class **ScoredGame** stores the score for each of the two teams. Make sure to include the necessary access methods. Consult the UML diagram above and the class **Run** below for the signatures of the methods.

```
public class Run {

    public static void main(String[] args) {

        Team senators, canadiens;

        senators = new Team("Ottawa Senators");
        canadiens = new Team("Montreal Canadiens");

        ScoredGame s;

        s = new ScoredGame(senators, canadiens);
        s.setScore(5, 0);

    }

}
```

A. Implement the class **Team** in the box below.

B. Implement the class **Game** in the box below.

C. Implement the class **ScoredGame** in the box below.

Question 2 (15 marks)

A. Fix 5 errors in the following Java program so that it compiles and runs without error. This method computes and returns the factorial of its argument. You can assume that the value of the parameter will always be greater than or equals to 0. Add the label of each of the five answers below (a, b, c, d, or e) onto the code, where the error is found. There could be more than one error per line of code.

```
public class Utils {

    public static factorial(n) {

        result = 1;

        for (i=1; i<=n; i++) {
            result = i * result;
        }

        result;
    }

}
```

(a) ____________________

(b) ____________________

(c) ____________________

(d) ____________________

(e) ____________________

B. In one or two sentences, explain why the following Java code does not compile. Circle the statement or statements that are the source of the problem.

```
Answer:
```

```
public class Person {
    private String name;
    private int age;
}
```

```
public class Child extends Person {
    private int grade;
    public Child(String name, int age, int grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
}
```

C. Given the declaration of the class **Cell** below.

```
public class Cell<E> {

    private E value;

    public Cell(E value) {
        this.value = value;
    }

    public boolean isEqual(Cell<E> other) {
        if (other == null) {
            return false;
        }

        if (this.value == null || other.value == null) {
            return this.value == null && other.value == null;
        }

        return this.value.equals(other.value);
    }
}
```

(a) Declare two reference variables, respectively named **s** and **t**, that can be used to designate objects of the class **Cell** to store **String** values.

(b) Give the Java statements to create two **Cell** objects to store **String** values, save the references these objects in the reference variables **s** and **t**. The first **Cell** object must be initialized with the value `"Suki"`, whereas the other must be initialized with `"Juliette"`.

(c) Give the Java expression for calling the method **isEqual** of the object designated by **s** passing the reference of the object designated by **t** as actual parameter.

(d) Declare an interface named **Taxable**. A **Taxable** object has a method **getTaxes** that returns a value of type **double**.

Question 3 (20 marks)

The interface **ExtendedStack** provides the usual stack methods, as well as the methods **count**, which returns the number of elements currently in the stack, and the method **reverse**, which reverses the elements in the stack.

The interface definition is provided below.

```
public interface ExtendedStack {
    boolean isEmpty();
    void push(String element);
    String pop();
    String peek();
    int count();
    void reverse();
}
```

Someone has partially implemented this interface in the class **ExtendedStackImplementation**. Specifically, methods **isEmtpy**, **push** and **pop** are already (correctly) implemented. **ExtendedStackImplementation** has only one constructor, with no parameter. This implementation can accommodate any number of elements.

Without making any assumptions on how **ExtendedStackImplementation** stores the stack (in particular, you cannot assume that the implementation uses an array) and using only the methods of **ExtendedStack** that have already been implemented (**isEmtpy**, **push** and **pop**), provide an implementation of the method **peek**. The method **peek** returns the top element of the stack without removing it. More precisely, after a call to the method **peek**, the content of the stack must be the same as it was before the call.

You don't need to handle the error situations, in particular, you can assume that **peek** will not be called if the stack is empty.

```
public class ExtendedStackImplementation implements ExtendedStack {

    // ...

    public String peek() {






    }

}
```

Still using only the methods already implemented, provide an implementation of the method **count**. The method returns the number of elements that are currently stored in the stack. The content of the stack must be the same before and after a call to **count**.

```
public class ExtendedStackImplementation implements ExtendedStack {

    // ...

    public int count(){










   }
}
```

Finally, provide the implementation of the method **reverse** which **reverses** the elements in the current stack (that is, it keeps the same elements, but stacked up in the reverse order).

```
public class ExtendedStackImplementation implements ExtendedStack {
...
    public void reverse(){




   }
}
```

Question 4 (20 marks)

Write a class to represent a permutation. Herein, a **permutation** is a rearrangement of the integers 0 to $n-1$, where n is the size of the permutation. The permutation that consists of the integers 0 to $n-1$ in-order is called the **identity-permutation**. For instance, $0, 1, 2, 3, 4$ is the identity-permutation for size 5.

Write a class called **Permutation**. Add all the necessary instance variable(s).

A. Implement the constructor **Permutation(int size)**, which initializes this permutation to the identity-permutation. The parameter **size** is the size of the permutation.

B. Implement the method **int getSize()**, which returns the size of this permutation.

C. Implement the method **int get(int pos)**, which returns the element at position **pos** of this permutation. You can assume that **pos** is not larger than the size of the permutation.

D. Write the method **rotate(int n)**. The method rotates the elements of this permutation by **n** positions, shifting the elements to the right (see exemple of a call below).

E. Implement the method **shuffle** that randomly rearranges the elements of this permutation. Use your time wisely. You might want to come back to this question once you have completed the rest of the examination.

Here is a sample use of the class **Permutation**:

```
Permutation p;
p = new Permutation(5);

for(int i = 0; i < p.getSize(); i++)
  System.out.print(" p[" + i + "] = "+ p.get(i));

p.rotate(2);

System.out.println("\n after rotation of 2:");
for(int i = 0; i < p.getSize(); i++)
  System.out.print(" p[" + i + "] = "+ p.get(i));
```

The code above produces the following output:

```
p[0] = 0 p[1] = 1 p[2] = 2 p[3] = 3 p[4] = 4
 after rotation of 2:
p[0] = 3 p[1] = 4 p[2] = 0 p[3] = 1 p[4] = 2
```

```
import java.util.Random;
```

(blank space)