0. Name (2 pts):

CS 2334: Programming Structures and Abstractions

Midterm Exam

Solution Set

Monday, October 5, 2009

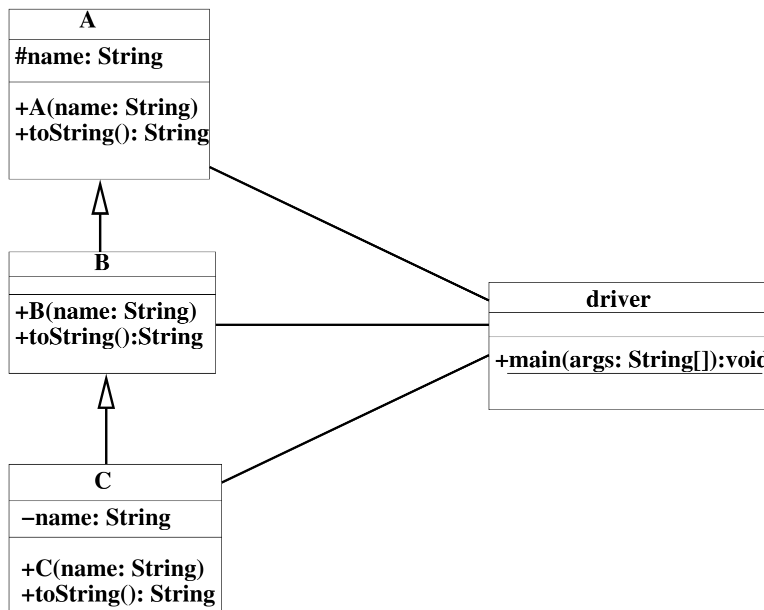
Problem	Topic	Max	Grade
0	Name	2	
1	Inheritance	30	
2	Abstract Classes and Interfaces	30	
3	Generic Programming and Generics	25	
4	Abstract Data Types	15	
Total			

1. **Inheritance** (30 pts)

Consider the following definition of four classes:

```
public class A
{
    protected String name;

    public A(String name){
        this.name = name;
    }

    public String toString(){
        return("A: " + name);
    };
}

public class B extends A
{
    public B(String name) {
        super("SUPER-B");
        this.name = name;
    }

    public String toString() {
        return("B: " + name);
    };
}

public class C extends B
{
    private String name;

    public C(String name){
        super("SUPER-C");
        this.name = name;
    };

    public String toString() {
        return("C: " + super.name);
    };
};

public class driver
{
    public static void main(String args[]) {
        A[] objects = new A[4];

        objects[0] = new A("foo");
        objects[1] = new B("bar");
        objects[2] = new C("baz");

        for(int i = 0; i < objects.length; ++i) {
            System.out.println(objects[i]);
        };
    };
};
```

(a) (15 pts) Draw the corresponding UML diagram. Include all variables, methods and relevant relations.



(b) (15 pts) What output does executing the driver class produce?

```
A: foo
B: bar
C: SUPER-C
null
```

Note: if you give any reasonable answer that indicates that *objects*[3] is null, then you will receive full credit.

2. Abstract Classes and Interfaces (30 pts)

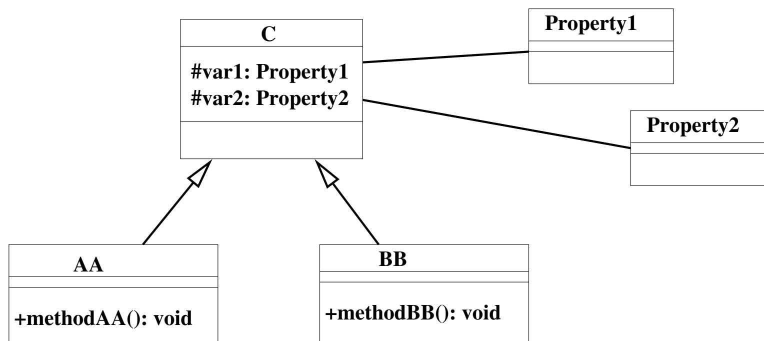
(a) (10 pts) Two classes, AA and BB share a common set of properties:

```
Property1 var1;
Property2 var2;
```

where **Property?** are other classes. However, there is one method defined for each class:

```
methodAA();    // For class AA
methodBB();    // For class BB
```

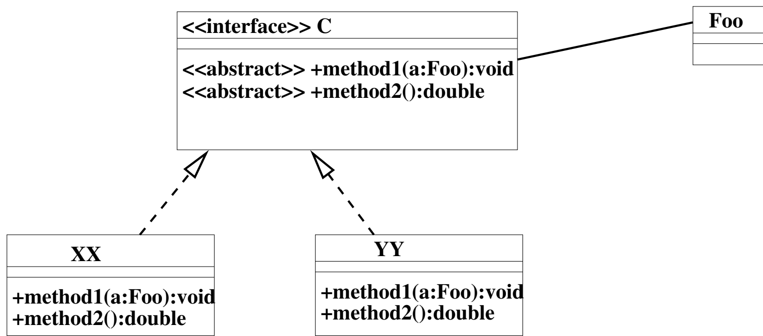
Draw the UML diagram that expresses the appropriate relationships between these classes. Include all relevant classes (including any that you need to invent), properties and methods.



(b) (10 pts) Two classes, XX and YY share a common set of method signatures:

```
void method1(Foo a);
double method2();
```

where **Foo** is a class. The method implementations and property sets are different for the two classes. Draw the UML diagram that expresses the appropriate relationships between these classes. Include all relevant classes (including any that you need to invent), properties and methods.



Note: if you implemented the common class as an abstract class, then I accept the solution as well.

(c) (10 pts) Briefly describe the conditions under which you would implement an abstract class in place of an interface.

The child class should have an *is-a* relationship with the parent. In addition, the parent can define properties and concrete methods.

3. **Generic Programming and Generics** (25 pts)

Assume the following initialization of two instances of our GenericQueue that we developed in class:

```
GenericQueue<Number> queue1 = new GenericQueue<Number>(10);
GenericQueue<Integer> queue2 = new GenericQueue<Integer>(10);
```

(a) (15 pts) Indicate whether the Java compiler will accept each of the following lines. Briefly explain why or why not.

```
queue1.add(new Integer(5));     // Accept: implicit cast from Integer
                //   to Number


queue2.add(4.7);  // Not accept: 4.7 is not of type Integer


queue1.add("foo");  // Not accept: "foo" is not of type Number


queue2.add(3);  // Accept: 3 will be autoboxed as an Integer


queue1.add(queue2.remove());  // Accept: an Integer is automatically
                  //   cast as a Number

.
```

(b) (10 pts) True or False and explain: generic programming requires the use of **generics**.

False. Generic programming is an approach of writing code that will accept many different types as input, which facilitates the re-use of code. *Generics* are not required for this in Java. However, they do help to enforce the proper implementation and use of generic code.

4. **Abstract Data Types** (15 pts)

The **GenericQueue** that we implemented in class captures the notion of a "line" of objects: new objects are inserted at the end of the line and objects are removed from the beginning of the line. A *deque* stands for a "double ended queue" in which new objects can be added to either the end **or** beginning of the line. Furthermore, removed objects can come from either the end or beginning of the line.

As a reminder, here are the properties of GenericQueue (note that they are now *protected*):

```
public class GenericQueue<T>
{
    protected T list[];
    protected int front;     // Next object to return
    protected int back;      // Next slot to insert a new object

    :
```

Fill in the requested method implementation below.
Hints: the value of $-1\%N$ is -1. **GenericQueue.getNum()** returns the number of objects currently in the queue.

```
public class GenericDeque<T> extends GenericQueue<T>
{
    public GenericDeque(int size) {
        super(size);
    };

    // Add obj to the front of the queue
    //
    // Return = true if successful addition
    //        = false if the queue is full
    //
    // Post: If queue is not already full:
    //      1. Size is increased by one
    //  2.  obj is in the array element indicated by front
    //

    public boolean addFront(T obj) {
        // Is the queue full?
        if(getNum() == list.length - 1)
            return(false);

        // No: move front backwards
        front = (front - 1 + list.length) % list.length;
        // Add the element
        list[front] = obj;

        return(true);
    }
}
```