0. Name (2 pts):

CS 2334: Programming Structures and Abstractions

Midterm Exam

Solution Set

Wednesday, October 6, 2010

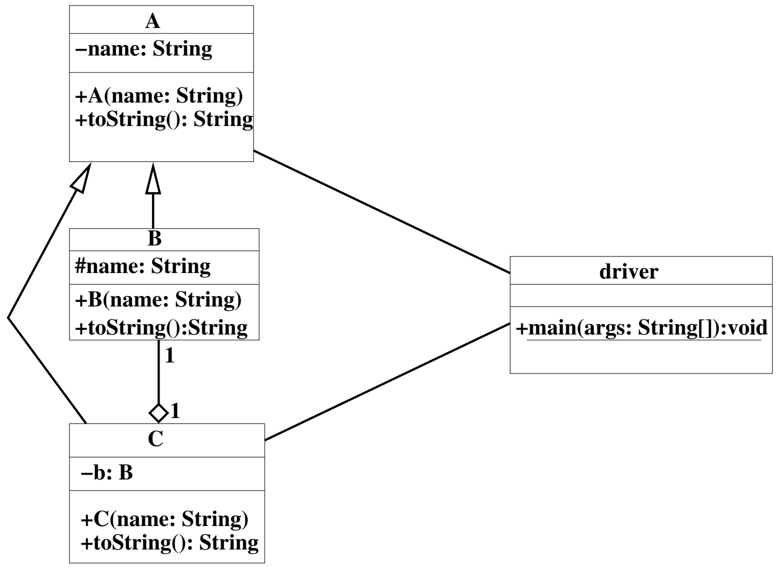
Problem	Topic	Max	Grade
0	Name	2	
1	Object Hierarchies	40	
2	Abstract Classes and Interfaces	25	
3	Generic Programming and Generics	15	
4	Abstract Data Types	20	
Total			

1. **Object Hierarchies** (40 pts)

Consider the following definition of four classes:

```
public class A
{
    private String name;

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

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

public class B extends A
{
    protected String name;

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

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

public class C extends A
{
    private B b;

    public C(String name){
        super(name);
        b = new B("subB");
    };

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

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

      objects[0] = new A("foo");
      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
null
C: A: baz, B: subB; A: SUPER-B
null
```

(c) (10 pts) For each line of code below, indicate one of the following:

- the compiler will reject the line (**REJECT**),
- the compiler will not reject the line but there will be a run-time error (**RUN-TIME**), or
- the compiler will not reject the line and there will be no run-time error (**OK**).

```
A a1 = new A("foo");      // ACCEPT: All A's are A's

C c = new C("baz");       // ACCEPT: All C's are C's

A a2 = new C("bar");      // ACCEPT: All C's are A's

B b2 = a1;                // REJECT: Not all A's are B's

B b3 = c;                 // REJECT: C's are not B's

C c3 = (C) a1;            // RUNTIME: This A is not a C (though
                             other A's could be)
```

2. **Abstract Classes and Interfaces** (25 pts)

(a) (10 pts) Briefly explain the distinction between abstract methods defined by an abstract class and the methods defined by an interface.

The answer I was looking for: There is no distinction. *Both types of abstract methods are contracts for what child classes must implement.*

There was one very technical answer that I also accepted: interface methods cannot be of default/protected visibility (whereas abstract methods can be). *Interfaces are all about making guarantees about what methods are provided to other classes. It therefore only makes sense for the methods to be declared as public*

(b) (5 pts) True or False and briefly explain: a method defined by an abstract class must be implemented by a child of that class.

False. However, if the child does not implement an abstract method, then is must also be declared as an abstract class.

(c) (10 pts) Give two reasons why *Comparable* is an interface and not an abstract class.

i. Does not define (or need to define) any instance variables.
ii. Many different types of objects should be Comparable, even though they may have no other similarities.
iii. Because a class in Java may only extend one other class, making Comparable an abstract class would severely limit the possible relationships with other classes.
iv. Comparable represents a *weak is-a (or is-kind-of-a) relationship*, so it does not warrant becoming an abstract class.

3. **Generic Programming and Generics** (15 pts)

Consider the following method prototypes and variable definitions:

```
public static <T> boolean find1(GenericQueue<T> q, T key);

GenericQueue<Number> a1 = new GenericQueue<Number>(5);
GenericQueue<Integer> a2 = new GenericQueue<Integer>(10);

Number v1 = new Integer(42);
Integer v2 = new Integer(24);
```

(a) (10 pts) Indicate whether the Java compiler will **REJECT** or **ACCEPT** each of the following lines. Briefly explain why or why not.

```
find1(a1, v2);   // Accept: implicit upcast of v2 to Number


find1(a2, v1);   // Not accept: not all numbers are integers


find1(a2, v2);   // Accept: all Integers are Integers

.
```

(b) (5 pts) True or False: The type parameters for generic classes are **only** checked at compile time.

True. *Type parameters are lost during the compilation process and hence cannot be checked at runtime (see the "type erasure" discussion).*

4. **Abstract Data Types** (20 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*):

```java
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.

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

    // Remove obj from the end of the queue
    //
    // Return = null if the queue is empty
    //        = the object at the end of the queue
    //
    // Post: If queue has an object in it, then:
    //       1. The number of objects in the queue is decreased by one
    //   2. The element at the back of the queue is removed
    //

    public T removeBack() {
          // Is the queue empty?
          if(isEmpty()) return null;

          // Move the back pointer: remember that 'back' is the
          //  *next* free space
          back = (back - 1 + list.length) % list.length;

          // Return the object at the back
          return(list[back]);
    }
}
```

Note: the above implementation has been changed to reflect the changes made on the board during the exam.