

CSE 120
Midterm II — March 4, 2009
ANSWERS

1. In this problem, you will add two new methods to the `CharSeqArrImp` class which implements the `CharSeq` interface presented. These were discussed in lecture; code is included in the code handout, modified to include additions to the `CharSeq` interface for the methods you will add. Neither of the methods you will provide depends on the other. Here are some interactions with the methods you will provide:

```
> char[] a = {'a','b','a','b','c'}
> CharSeq s = CharSeqArrImp.makeSeq(a);
> s
a b a b c
> CharSeq w = CharSeqArrImp.makeSeq(new char[] {'a','b'})
> w
a b
> s.findNth(w,0)
0
> s.findNth(w,1)
2
> s.findNth(w,2)
-1
> s
a b a b c
> s.rotate(1)
> s
b a b c a
> s.rotate(0)
> s
b a b c a
> s.rotate(3)
> s
c a b a b
>
```

- (a) (8 points) Write below a method `findNth`

```
public int findNth(IntSeq word, int n).
```

This method is intended to return the index of the n th occurrence of a given word, counting from 0, or -1 if the word does not occur at least $n + 1$ times in this sequence. (Hint: reuse the code for `find`, modified appropriately.)

Answer:

```
public int findNth(CharSeq word,int n) {
    int where = -1;
    int count = 0;
    for (int pos = 0; pos <= size(); pos++) {
        if (occursAt(word, pos)) {
            if (count++ == n) {
                where = pos;
                break;
            }
        }
    }
    return where;
}
```

- (b) (8 points) Write below a method `rotate`

```
public void rotate(int n)
```

This method rotates the sequence it is invoked on n positions “counterclockwise”. That is, if n is 1, the 0th element becomes the last element, the 1st element becomes the 0th element, and in general the i th element becomes the $i - 1$ element. The method should have no effect on a sequence with fewer than two elements.

Answer:

```
public void rotate(int n) {
    if (size() < 2) return;
    for(int j = 0; j < n; j++) {
        char tmp = data[0];
        for (int i = 1; i < size(); i++) {
            data[i-1] = data[i];
        }
        data[size()-1] = tmp;
    }
}
```

Grading guidelines:

For both parts

- 8 points Correct logic, works for boundary cases.
- -2 points ccorrect logic, minor boundary case errors.
- -1 point For array index out of bounds and similar errors.
- 0 points Completely incorrect, incomplete answers.

Some comments about specific errors

- Part a: -2 points If $(n-1)$ th occurrence position is returned instead of n th.
- Part a: -4 to -6 points Partially correct code- returns correct position in some cases.
- Part b: -1 points Data is final, trying to assign a temp array to data.
- Part b: -2 points Does not rotate n times.
- Part b: -4 points Elements partially rotated, incorrect output.

2. There are three *compilation* errors in the following code fragment. Circle each error, mark it with a letter (a)-(c), and in the space below, *briefly* describe what caused the error. There is only one error per line. (Don't waste time trying to figure out what this program does!)

(3 points for each error found. 9 points total)

```
abstract class FunCountImp implements FunnyCount {
    private int count = 512;
    public abstract int getCount();
    public void countExceeded() { throw new CountExceededException(); }
    public void maybeIncrement() {
        FunnyCount x = this;
        if (x.getCount() >= 10) {
            return;
        }
        incr();
    }

    public static void incr() { this.count++;}
}

public class CountExceededException extends Exception{
    public CountExceededException(String s){super(s);}
    public CountExceededException(){super();}
}

interface FunnyCount {
    public void countExceeded();
}
```

Answer:

The three errors are:

- The method `countExceeded` fails to declare that it throws `CountExceededException`.
- The static method `incr` cannot access the dynamic instance variable `count`.
- The local variable `x` in the method `maybeIncrement` cannot call the method `getCount` because its static type is `FunnyCount`.

Grading scheme:

- -2 points for correct error location but incorrect reasoning
- -3 points for incorrect error location

3. This problem concerns the two Java source files, **TicketFactory.java** and **Ticket.java**, found in the code handout. Please take a minute to look over these source files.

Your job is to implement the class **OnePerFactory** that adapts the **TicketFactory** class so that a single person can buy at most *one* ticket from a *particular* ticket factory.

A sample interaction with this new class might be as follows:

```
> import ticket.*;
> OnePerFactory tf = new OnePerFactory("Springsteen", 200);
> Ticket t0 = tf.issue("Victoria");
> t0
Ticket # 0 issued to Victoria
> Ticket t1 = tf.issue("Annie");
> t1
Ticket # 1 issued to Annie
> Ticket t2 = tf.issue("Victoria");
> t2
null
```

The beginning of the definition of class **OnePerFactory** is below. This definition includes the declaration of two instance variables: **tickets** should store an array of tickets issued by the factory, and **ticketFactory** is an instance of the **TicketFactory** class. Your task will be to implement the constructor and two methods for this class. You may not add any new instance variables, nor change the declarations below.

```
package ticket;

public class OnePerFactory {
    // Array of tickets that have been issued by this factory.
    private Ticket[] tickets;

    // A ticket factory
    private TicketFactory ticketFactory;
```

- (a) (3 points) Complete the constructor for this class.

Answer:

```
public OnePerFactory(String name, int maxTickets) {
    tickets = new Ticket[maxTickets];
    ticketFactory = new TicketFactory(name, maxTickets);
}
```

- (b) (4 points) Implement a helper method of the class **OnePerFactory** that determines whether a person has already bought a ticket. *Answer:*

```
private boolean bought(String purchaser) {

    for (int i=0; i<ticketFactory.getIssued(); i++) {
        if (tickets[i].getPurchaser().equals(name)) {
            return true;
        }
    }
    return false;
}
```

- (c) (6 points) Implement the *issue* method to make sure that each purchaser can only buy one ticket. If a purchaser has previously bought a ticket, this method should return `null`.

Answer:

```
public Ticket issue(String purchaser) {  
  
    if (bought(purchaser))  
        return null;  
    else {  
        Ticket t = ticketFactory.issue(purchaser);  
        if (t == null) {  
            return null;  
        }  
        tickets[ticketFactory.getIssued()-1] = t;  
        return t;  
    }  
}
```

Grading scheme:

- Part a
 - -1 point for not initializing `tickets[]`.
 - -2 points for not initializing `ticketFactory`.
- Part b
 - -1 if the method throws a `NullPointerException`.
 - -1 for accessing private members of class `TicketFactory`.
- Part c
 - -1 for Accessing private members of class `TicketFactory`.
 - -1 for not returning a `Ticket`
 - -1 for not using `ticketFactory.issue`.
 - -2 for not updating `tickets[]`.

4. (12 points) The next problem concerns the implementation of the method `divide`.

The static method, `divide` in the class `IntegerDivision`, prompts the user for two integers, divides the first integer (the dividend) by the second (the divisor) using integer division and returns the result. If the divisor is zero, and hence the computer attempts to divide by zero, the method should prompt the user for another two numbers. For example, a DrJava interaction might look like:

```
> IntegerDivision.divide(System.in)
Enter dividend:
 5
Enter divisor:
 0
Divide by zero - try again
Enter dividend:
 5
Enter dividend:
 2
2
>
```

In the implementation of `divide`, all input should be from the parameter `input` of type `InputStream`, and all output should be to `System.out`, using the method `println`. Input should be processed using an instance of the class `java.util.Scanner` created in the following manner:

```
Scanner scanner = new Scanner(input);
```

To read inputs, `divide` should use the method `nextInt` from the `java.util.Scanner` class. More information about this method is available in the Appendix.

The `divide` method must be very careful about exceptions during input. If the user inputs a zero as the divisor, `divide` will attempt to divide by zero, and throw an `ArithmeticException`. In this case, `divide` should catch the exception and ask the user for other inputs, as shown above. If the user inputs are not integers, `nextInt` will throw a `InputMismatchException`. If other input errors occur, `nextInt` will throw two other runtime exceptions (see the documentation in the code appendix). If any of these three exceptions are thrown, `divide` should catch them and throw a `ScanningException` (which should extend `Exception`) to indicate that there was some trouble. You may ignore exceptions that could arise from `System.out.println`.

In the class `IntegerDivision` on the next page, define the method `divide`, which takes a single argument, the `InputStream` `input`, and returns an `int`. Do not forget to include the appropriate `throws` annotation.

Answer:

```
import java.io.*;
import java.util.*;

public class IntegerDivision {

    static class ScanningException extends Exception {}

    public static int divide() throws ScanningException {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            try {
                System.out.println("Enter dividend:");
                int dividend = scanner.nextInt();
                System.out.println("Enter divisor:");
                int divisor = scanner.nextInt();
                return dividend/divisor;
            } catch (ArithmeticException e){
                System.out.println("Divide by zero - try again");
            } catch (InputMismatchException e){
                throw new ScanningException();
            } catch (NoSuchElementException e) {
                throw new ScanningException();
            } catch (IllegalStateException e){
                throw new ScanningException();
            }
        }
    }
}
```

Grading guidelines:

- 1 point Declaring that the divide method throws ScanningException.
- 1 point Declaring that the divide method is static.
- 2 points Code for ScanningException class.
- 1 point Creating a Scanner obj from the passed InputStream object.
- 1 point Code for getting user input.
- 1 point Enclosing the appropriate statements in 'try' block.
- 2 points Handling the 3 exceptions thrown by scannerobj.nextInt() by throwing a ScanningException.
- 2 points Handling the ArithmeticException and code prompting the user to reenter the numbers.
- 1 point Proper syntax and clear code.