CIS 120 Final Exam December 17, 2012

SOLUTIONS

1. True or False (20 points)

All of the following questions refer to Java programming.

a. $\boxed{\text{T}}$ F Whenever you override the `equals` method of a class, you should be sure to override the `hashCode` method compatibly.

b. T $\boxed{\text{F}}$ If `A` is a subtype of `B`, then `Set<A>` is a subtype of `Set<B>`.

c. $\boxed{\text{T}}$ F When using the Java IO libraries, one should generally wrap a `FileReader` object inside a `BufferedReader` to prevent significant performance problems.

d. T $\boxed{\text{F}}$ It is possible to create an object of type `Set<int>`.

e. T $\boxed{\text{F}}$ An object's *static* type is always a subtype of its *dynamic* type.

f. T $\boxed{\text{F}}$ The value `null` can be assigned to a variable of any type.

g. $\boxed{\text{T}}$ F A method with the following declaration definitely *will not* throw an `IOException` to a calling context.

```
public void m() {...}
```

h. $\boxed{\text{T}}$ F The `@Override` annotation prevents accidental overloading of a method.

i. T $\boxed{\text{F}}$ In some cases, dynamic dispatch of a method invocation requires the Java Abstract Stack machine to search the entire stack to find the appropriate code to run next.

j. T $\boxed{\text{F}}$ It is not possible to call a method declared as `static` from within a non-`static` method.

Rubric: 2 points apiece.

2. Java Design Problem (30 points)

A *stack* is a collection (like a list or a set) that provides restricted access to an ordered sequence of elements. In this problem, you will apply the design process to implement a linked stack structure; in Problem 3 you will write a program that uses the stack abstraction.

Step 1: Understand the problem. A stack is a collection that offers access only to the *top* element of an ordered sequence. Think of a stack as a "pile" of objects: you can add something to the top of the stack or take something off the top, but you can't get to a lower object without removing the ones above it first. We saw one use of a stack in the *abstract stack machine* in which the elements of the stack were the variable-value binding pairs. Here we will develop a generic stack collection. (There is nothing to do for this step, your understanding will be demonstrated below.)

Step 2: Design the interface. Stacks offer three operations: You can *push* an element on to the top the stack, you can *pop* an element from the top of the stack (leaving the rest of the elements alone), or you can ask whether the stack *is empty*. We thus represent the stack interface in Java like this:

```java
public interface Stack<A> {
    // test whether the stack has any elements
    boolean isEmpty();

    // adds elt to the top of the stack
    void push(A elt);

    // returns the top of the stack (if any)
    // throws NoSuchElementException of the stack is empty
    A pop();
}
```

a. (2 points) Suppose you were going to use the `java.util` standard libraries. Which of the following types could most naturally be used as the underlying representation of a stack in a class that implements the interface `Stack<A>`? (Choose one answer.)

- `HashSet<A>`
- $\boxed{\texttt{LinkedList<A>}}$
- `TreeMap<A,A>`
- `Object[]`
- `Queue<Integer>`

b. $\boxed{\text{T}}$ F (2 points) It is possible to create a class that implements both the `Stack<A>` and `Set<A>` interfaces. (true or false)

c. $\boxed{\text{T}}$ F (2 points) It is possible to create a class that implements both the `Stack<A>` and `Set<Integer>` interfaces. (true or false)

In the rest of this problem, you will implement the `Stack<A>` interface *without* using the `java.util` libraries. The class is called `LinkedStack<A>`, and it will use a singly-linked, mutable data structure similar to the `queue` and `deque` types from homework 5.

Step 3: Write test cases. A stack is supposed to exhibit "last in, first out" (LIFO) behavior: the most recent element pushed is the next one that will be popped. Complete the following test cases that demonstrate this behavior. We have also provided two complete example tests.

Rubric: 1 point per blank (6 points total)

```
import ...;
import java.util.NoSuchElementException;

public class LinkedStackTest {
      @Test
      public void testIsEmpty() {
            LinkedStack<Integer> s = new LinkedStack<Integer>();
            assertEquals(true, s.isEmpty());
      }

      @Test
      public void testPopEmptyException() {
            LinkedStack<Integer> s = new LinkedStack<Integer>();
            try {
                  s.pop();
                  Assert.fail();
            } catch (NoSuchElementException e) { // expected outcome
            }
      }
// ------------------ FILL IN HOLES BELOW -------------------------
      @Test
      public void testPushIsNotEmpty() {
            LinkedStack<Integer> s = new LinkedStack<Integer>();
            s.push(3);
            assertEquals(false, s.isEmpty());
      }

      @Test
      public void testPushPushPopPop() {
            LinkedStack<Integer> s = new LinkedStack<Integer>();
            s.push(3);
            s.push(4);
            assertEquals((Integer) 4, s.pop());
            assertEquals((Integer) 3, s.pop());
            assertEquals(true, s.isEmpty());
      }

}
```

Step 4: Implement it. Implement the `LinkedStack<A>` class. Use an auxiliary inner class called `Node` to store the (singly-linked, mutable) stack structure—it doesn't need any methods. ***Do not use any library classes***; do *not* use arrays; you should not need to catch any exceptions. The `LinkedStack<A>` default constructor is sufficient. Your code should pass the tests given above.

```
import java.util.NoSuchElementException;

public class LinkedStack<A> implements Stack<A> {
	// field(s) of the LinkedStack<A> class
	private Node top;

	// private class of Node data
	private class Node {
		// field(s) of the Node class
		A elt;
		Node next;

		// constructor of the Node class
		Node(A elt, Node next) {
			this.elt = elt;
			this.next = next;
		}
	}

	// returns true if the Stack is empty
	public boolean isEmpty() {
		return (top == null);
	}

	// pushes an element onto the top of the stack
	public void push(A elt) {
		Node newTop = new Node(elt, top);
		top = newTop;
	}

	// pops the top element off the stack, updating
	// the stack in place
	// throws a NoSuchElementException if the stack is empty
	public A pop() {
		if (this.isEmpty())
			throw new NoSuchElementException();
		Node t = top;
		top = t.next;
		return t.elt;
	}
}
```

Rubric:

- 2 points for private field (1 pt for private, 1 pt for field)
- 4 points for node class (1 pt for each field, 2 pts for correct constructor)
- 2 points for `isEmpty`

- 4 points for `push` (2 pts for new node, 2 pts for reassign)
- 6 points for `pop` (2 pts for exception, 2 pts for top/t.next , 2 pts for returning `t.elt` not `t`)

3. Java Programming (20 points)

In this problem, you will write a Java program that uses the `Stack<A>` abstraction from Problem 2. Consider the problem of *matching well-nested brackets* when reading characters from some data source. For example, you might want to determine whether each open parenthesis character `'('` is later followed by a matching `')'`, and similarly for `'{'` and `'}'` and `'['` and `']'`. Eclipse uses such an algorithm to check that a Java program doesn't have syntax errors.

One subtlety is that such brackets should not only match, but they should also be *well-nested*. The string `"( [ ) ]"` is *not* well-nested because the `[` is closed by `)` and not `]`, as it should be. On the other hand, the string `"()[()]"` is well-nested.

We have written the following test cases that illustrate many more examples of the desired behavior of this method, called `BracketMatcher.matched`. For the purposes of this problem, the matcher simply ignores non-bracket characters. Be sure you understand these tests before continuing.

```
import static org.junit.Assert.*;
import java.io.Reader;
import java.io.StringReader;
import org.junit.Test;

public class BracketMatcherTest {
      private void testString(boolean matched, String s) {
            Reader r = new StringReader(s);
            assertEquals(matched, BracketMatcher.matched(r));
      }

      @Test
      public void testMatchesEmpty() {
            testString(true, "");
      }

      @Test
      public void testNoMatchOpen() {
            testString(false, "(");
      }

      @Test
      public void testNoMatchClose() {
            testString(false, ")");
      }

      @Test
      public void testMatchOpenClose() {
            testString(true, "()");
      }

      @Test
      public void testNoMatchOpenOpenClose() {
            testString(false, "(()");
      }

      @Test
      public void testMatchOpenOpenCloseClose() {
            testString(true, "(())");
      }
```

```
    @Test
    public void testMatchOpenOpenCloseCloseJunk() {
          testString(true, "(x(e)xxabxx)");
    }

    @Test
    public void testMatchOpen1Open2Close2Close1() {
          testString(true, "([])");
    }

    @Test
    public void testNoMatchOpen1Open2Close1Close2() {
          testString(false, "([)]");
    }

    @Test
    public void testMatchProblemStatement() {
          testString(true, "()[()]");
    }

    @Test
    public void testNoMatchOpen1Open2Close1Close2Junk() {
          testString(false, "(ax[asda)b]");
    }

    @Test
    public void testMatchOpen3Open2Open1Close1Close2Close3() {
          testString(true, "{[()]}");
    }
}
```

How do we implement such a matching algorithm? We read through the sequence of characters (provided by a `Reader` object as in the Spellchecking project). Each time we encounter a left bracket (like `'['`), we *push* it onto a stack. Each time we encounter a right-bracket (like `']'`), we pop the top of the stack and check to make sure that it matches. If we ever hit a mismatch, or if the stack is empty when it shouldn't be, then the sequence isn't well-nested. After reading all of the sequence, if the stack isn't empty, the sequence isn't well matched.

Implement this algorithm on the following page. We have provided a couple utility methods that may be of use. Your code should pass all of the tests above and it should never raise any exceptions.

- For your convenience, the documentation of the `read()` method of the `Reader` class is found in the Appendix — you should not need to use any other methods from the reader class.
- You may need to use the `intValue()` method that is provided by the `Integer` class. It returns the **`int`** underlying a given `Integer` object. For example, `(new Integer(3)).intValue()` evaluates to `3`.

```java
import java.io.IOException;
import java.io.Reader;
public class BracketMatcher {

        // returns true if c is a legal left bracket
        private static boolean isLeftBracket(int c) {
                return c == '(' || c == '{' || c == '[';
        }

        // returns true if c is a legal right bracket
        private static boolean isRightBracket(int c) {
                return c == ')' || c == '}' || c == ']';
        }

        // returns true if l and r are a matched bracket pair
        private static boolean isBracketPair(int l, int r) {
                return (l == '(' && r == ')')
                                ||
                                (l == '{' && r == '}')
                                ||
                                (l == '[' && r == ']');
        }

        // Returns true if the sequence of brackets read from r form well-nested matching pairs.
        // Ignores non-bracket characters.
        public static boolean matched(Reader r) {
                if (r == null) return false; // OPTIONAL
                Stack<Integer> s = new LinkedStack<Integer>();
                try {
                        int c = r.read();
                        while (c != -1) {
                                if (isLeftBracket(c))
                                        s.push(c);
                                if (isRightBracket(c)) {
                                        if (s.isEmpty()) {
                                                return false;
                                        } else {
                                                int d = s.pop().intValue();
                                                if (!isBracketPair(d, c))
                                                        return false;
                                        }
                                }
                                c = r.read();
                        }
                } catch (IOException e) {
                        return false;
                }
                if (s.isEmpty())
                        return true;
                return false;
        }
}
```

Rubric:

- Instantiating `Stack` (1 pt. typing, 1 pt. constructor)
- Reading chars: 1 pt. read before first check, 1 pt. update around loop
- While loop: 2 pts. correct condition
- Try/Catch: exists 1 pt., correct exception 1 pt.
- Dealing with left, right bracket conditions: 1 pt.
- Push: 2 pt.
- Empty case: 2 pt.
- pop: 2 pt., use of `intValue()` 1 pt.
- correct call to `isBracketPair` 2 pt.
- empty at the end – correct return: 2 pt.

4. Abstract Stack Machine (10 points)

Consider the following Java class (which is similar to the many linked datastructures we've seen in class):

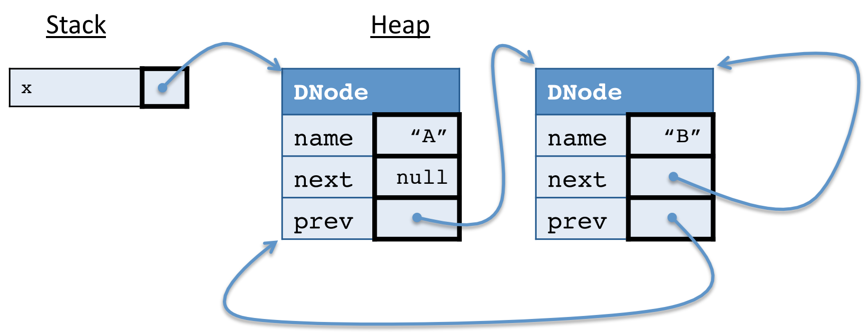
```
public class DNode {

      String name;
      DNode next;
      DNode prev;

      DNode(String name, DNode n, DNode p) {
            this.name = name;
            this.next = n;
            this.prev = p;
      }

      void foo() { ... }
}
```

Suppose that, for some inscrutable reason, you wanted to write a program to construct the following stack and heap configuration:



Write a sequence of Java commands for the body of method `foo` that would create the ASM configuration shown above (ignore the workspace that would be saved on the stack by a call to `foo()`). Note that there is only *one* variable on the stack.

Rubric;

- 4 points for constructors: (1 points for both use of **new** `DNode` constructor, 1 point for correct arguments)
- 3 points for "wiring": 1 point per pointer
- 3 points for stack shape: 1 point for type declaration, 2 points for only using variable `x`

Two example solutions (there are many others):

```
DNode x = new DNode("B", null, null);
x.next = x;
x = new DNode("A", null, x);
x.prev.prev = x;
```

or

```
DNode x = new DNode("A", null, null);
x = new DNode("B", null, x);
x.next = x;
x.prev.prev = x;
x = x.prev;
```

5. Java's Type System (8 points)

Consider the following class definitions:

```
class S {
}

class T extends S {
}

class U extends S {
}

class V extends U {
}
```

For each of the questions below, circle *all* the correct answers—there may be zero, one, or more.

a. What is the *static* type of `x` in the code below?

```
S x = new V();
x = new U();
```

a. `Object` b. $\boxed{\texttt{S}}$ c. `T` d. `U` e. `V`

b. What is the *dynamic* type of the value stored in `x` after running the code below?

```
S x = new V();
x = new U();
```

a. `Object` b. `S` c. `T` d. $\boxed{\texttt{U}}$ e. `V`

c. Which types can we place in the hole marked __?__ below so that no `ClassCastException` is thrown when this program is run?

```
Object o = new U();
Object x = (__?__)o;
```

a. $\boxed{\texttt{Object}}$ b. $\boxed{\texttt{S}}$ c. `T` d. $\boxed{\texttt{U}}$ e. `V`

d. Which types, when placed in the hole marked __?__ below, cause the compiler to generate an "incompatible types" error message?

```
T t = new T();
boolean b = t instanceof __?__;
```

a. `Object` b. `S` c. `T` d. $\boxed{\texttt{U}}$ e. $\boxed{\texttt{V}}$

Rubric:

- 1 pt. for each of the seven circles.
- no extra circles allowed in first two parts
- 8th point if no extra circles in last two parts
- -1 point for each additional circle after 2

6. **First-class Functions and fold** (12 points)

Recall the OCaml definition of binary trees, and consider the `fold` function for such trees:

```
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let rec fold (combine : 'a -> 'b -> 'b -> 'b) (base : 'b) (t : 'a tree) : 'b =
 begin match t with
   | Empty -> base
   | Node(lt, x, rt) -> combine x (fold combine base lt) (fold combine base rt)
 end
```

In this problem, you will explain how many functions can be written in terms of `fold`. Consider the following recursive tree functions:

```
let rec sum (t : int tree) : int =
 begin match t with
   | Empty -> 0
   | Node(lt, x, rt) -> (sum lt) + x + (sum rt)
 end

let rec size (t : 'a tree) : int =
 begin match t with
   | Empty -> 0
   | Node(lt, _, rt) -> (size lt) + 1 + (size rt)
 end

let rec is_full (t : 'a tree) : bool =
 begin match t with
   | Empty -> true
   | Node(lt, _, rt) ->
       ((size lt) = (size rt)) && is_full lt && is_full rt
 end

let rec preorder (t : 'a tree) : 'a list =
 begin match t with
   | Empty -> []
   | Node(lt, x, rt) -> x::(preorder lt)@(preorder rt)
 end

let rec postorder (t : 'a tree) : 'a list =
 begin match t with
   | Empty -> []
   | Node(lt, x, rt) -> (postorder lt)@(postorder rt)@[x]
 end

let rec inorder (t : 'a tree) : 'a list =
 begin match t with
   | Empty -> []
   | Node(lt, x, rt) -> (inorder lt)@[x]@(inorder rt)
 end
```

For each function above, select the combination of `combine` and `base` arguments to `fold` such that you get an equivalent implementation by writing:

```
fun t -> fold combine base t
```

Rubric: 1 pt. per blank

	`combine`	`base`
`sum`	d	0
`size`	c	0
`is_full`	none	none
`preorder`	h or i	`[]`
`postorder`	f	`[]`
`inorder`	g	`[]`

Your choices for `combine` and `base` are enumerated below. Write the choice a–j for the `combine` function and the choice a–j for `base`. If no choice works (*i.e.* the function *cannot* be expressed as a fold) write "none" in both slots. You may use the same choice more than once.

Combine choices:		Base choices:	
(a)	`(fun x lv rv -> Node(rv, x, lv))`	(a)	**`true`**
(b)	`(fun x lv rv -> lv + rv)`	(b)	**`false`**
(c)	`(fun x lv rv -> lv + 1 + rv)`	(c)	`0`
(d)	`(fun x lv rv -> lv + x + rv)`	(d)	`1`
(e)	`(fun x lv rv -> (size lv) = (size rv))`	(e)	`Empty`
(f)	`(fun x lv rv -> lv@rv@[x])`	(f)	`[]`
(g)	`(fun x lv rv -> lv@[x]@rv)`	(g)	`[x]`
(h)	`(fun x lv rv -> [x]@lv@rv)`	(h)	`lv`
(i)	`(fun x lv rv -> x::lv@rv)`	(i)	`rv`
(j)	`(fun x lv rv -> lv@rv@x)`	(j)	`lv@rv`

7. **Binary Search Trees & OCaml programming** (20 points)

This problem uses the same OCaml type of trees as in Problem 6, repeated here for your reference:

```
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)
```

a. State the *binary search tree* invariant in words. We have done the case for `Empty` trees:

- The `Empty` tree is a binary search tree.
- The tree `Node(lt, x, rt)` is a binary search tree if and only if:
  `lt` and `rt` are both binary search trees and `x` is greater than all values in `lt` and less than all values in `rt`.
  Rubric: 5 pts.: 2 pts. for `lt` and `rt` are both bsts. 3 pts for less-than and greater-than constrainta

b. Write an OCaml function `range` that, given an integer binary search tree `t` and integers `low` and `hi` such that `low < hi`, returns the list of BST nodes such that `low <= x <= hi` (in sorted order). For example, `range t 1 6` would yield `[1;2;3;5]` when `t` is the tree to the right. Use the binary search tree invariant to avoid processing more of the tree than necessary. If you need help remembering OCaml syntax, see the examples in problem 6.

```
        3
       / \
      /   \
     1     8
    / \   /
   0   2 5
          \
           7
```

```
(* Assumes: t is a binary search tree *)
let rec range (t:int tree) (low:int) (hi:int) : int list =
  begin match t with
    | Empty -> []
    | Node(lt,x,rt) ->
      if low <= x && x <= hi
      then (range lt low hi) @ [x] @ (range rt low hi)
      else if x < low
      then range rt low hi
      else range lt low hi
  end
```

Rubric: 15 pts.

- 3 pts for base case
- 5 pts for `x` is in range and recursive calls
- 7 pts for using invariant to search when `x` is not in range

Reference Appendix

Make sure all of your answers are written in your exam booklet. These pages are provided for your reference—we will *not* grade any answers written in this section.

Reader JavaDoc (excerpt) for problem 3

From the `Reader` JavaDocs:

- **`int`** `java.io.Reader.read()` **`throws`** `IOException`

  Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

  Subclasses that intend to support efficient single-character input should override this method.

  Returns: The character read, as an integer in the range `0` to `65535` (`0x00`-`0xffff`), or `-1` if the end of the stream has been reached

  Throws: `IOException` — If an I/O error occurs