CIS 120 Final Exam December 16, 2013

SOLUTIONS

1. True or False (10 points)

a. [T] F In OCaml, the binding between a variable and its value can never be changed.

b. T [F] In OCaml, if `x` is a variable of any type, `Some x == Some x` will always return **`true`**.

c. [T] F In Java, it is good practice, when overriding the `equals` method from `Object`, to override the `hashCode` method as well.

d. T [F] In Java, the default implementation of equality in the `Object` class uses reference equality for mutable objects and structural equality for immutable objects.

e. T [F] Java's *subtype polymorphism* is very similar to OCaml's *parametric polymorphism*, whereas Java's *generics* are a different thing with no direct analog in OCaml.

f. [T] F In Java, the dynamic class of the result of an expression will always be the same as or a subtype of its static type, even if casts are used.

g. T [F] In a Java `try-catch-finally` statement, the `finally` clause is executed only if the `try` clause does not throw an exception.

h. [T] F In Java, a `final` instance variable can only be changed in a constructor.

i. T [F] In Java, every method must declare in its header every exception it might throw.

j. T [F] In Java, if type `A` is a subtype of `B`, then `Set<A>` is a subtype of `Set<B>`.

2. **Array processing** (20 points)

In this problem you will implement a static method called `isPermutation` that takes as input two arrays of integers and returns true if the second array is a *permutation* of the first — i.e., the two arrays contain exactly the same elements, though perhaps in different orders. For example, if

```
int[] a = { 1, 2, 3, 3, 3 };
int[] b = { 3, 1, 3, 2, 3 };
int[] c = { 1, 2, 3 };
int[] d = { 1, 2, 3, 4, 5 };
int[] e = { 1, 1, 1, 2, 3 };
```

then `isPermutation` should return **`true`** when called on `a` and `b` and **`false`** when called on any other pair.

Getting this exactly right is a bit trickier than it might seem at first. One way to approach it is to notice that, if the arrays have the same length, then one is a permutation of the other if and only if each distinct element appears the same number of times in both. Remember to do something reasonable on null inputs. Do not use any external libraries such as collection classes.

```
class Main {
  public static boolean isPermutation(int[] x, int[] y) {
    if (x == null || y == null) return (x == y);
    if (x.length != y.length) return false;
    for(int i=0; i<x.length; i++) {
      int c1 = 0;
      int c2 = 0;
      for(int j=0; j<x.length; j++) {
        if (x[j] == x[i]) c1++;
        if (y[j] == x[i]) c2++;
      }
      if (c1 != c2) return false;
    }
    return true;
  }
}
```

Grading Scheme: In general,

- *perfect (and only perfect) solutions got 20 points;*
- *mostly correct solutions got between 15 and 19 points;*
- *solutions with good ideas but significant problems got 9-14 points;*
- *solutions with more serious problems got 0-8 points.*

Some common points:

- *Solutions that didn't handle duplicate elements got at most 7 points.*
- *Solutions that would fail if the* `x` *array contains negative integers got at most 12 points.*
- *Knocking out already-found elements of* `y` *by setting them to* `null` *lost 5 points. Knocking out elements of* `y` *with zeroes or MAX_INTs lost 8 points.*

- *Solving the problem by sorting the arrays and then comparing but getting the sort function wrong got at most 10 points.*
- *Extremely inefficient solutions (e.g., allocating an array the size of the difference between the maximum and minimum elements of x) got 19 points if otherwise perfect.*
- *Solutions that used external libraries like HashMap got at most 10 points.*
- *Forgetting null check: -2. (We accepted pretty much any behavior on null inputs, as long as there was an explicit check.)*
- *Small errors in otherwise perfect solutions: -1 to -3*

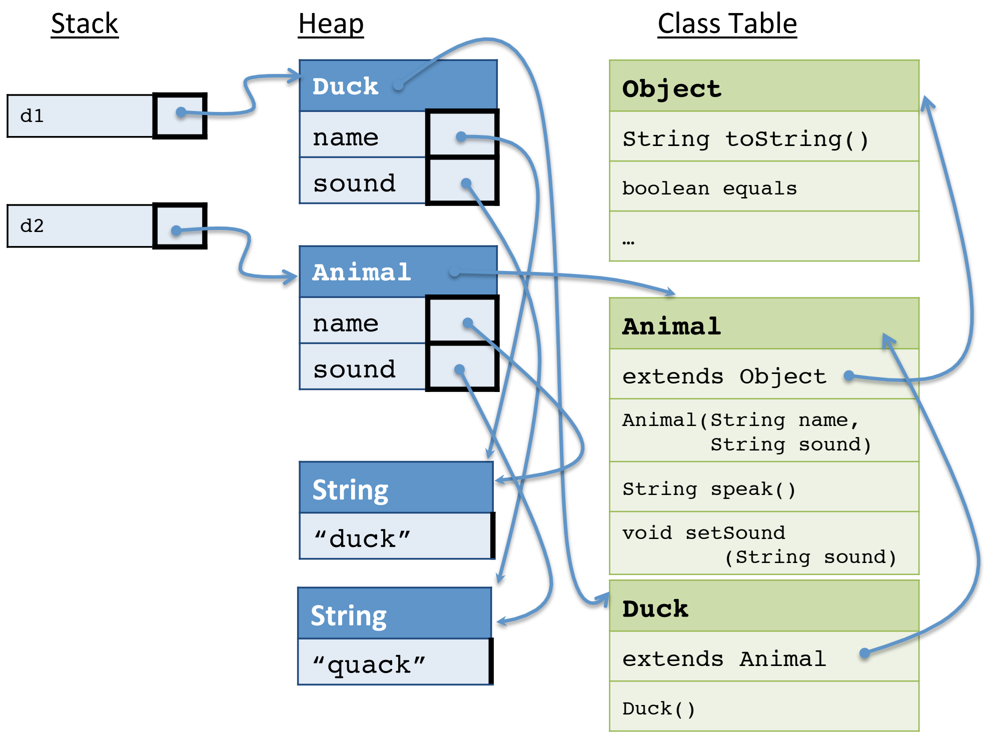
3. Java ASM (12 points)

On page 4 of the appendix there are some object definitions and a `main` method refering to those.

For the `String` objects, you do not need to draw the class table part. Follow the pattern of the appendix in showing the `String` objects in the heap.

Draw the Java ASM (including the stack, heap, and class table) at the point of the computation marked */* Here */*. Do not write out the code in the method bodies: just show the headers for the methods belonging to each class in the class table.

Answer:



Grading Scheme: 2 pts for having d1 and d2 pointers. 3 pts each (6 pts total) for having the proper heap objects.

- *1 pt for the box existing with proper structure (names of fields).*
- *1 pt for the box having the correct class label/pointer.*
- *1 pt for having name and sound pointing to strings in the heap.*

4 pts for having the correct class table.

- *1 pt for having Object in the class table.*
- *1 pt for having Animal in the class table.*
- *1 pt for having all the correct methods of Animal in Animal's entry.*

- *1 pt for having Duck in the class table that extends Animal.*

We did not take off points for drawing four string objects in the heap rather than two. The actual correct answer is two (the Java compiler shares statically-known `String` *objects), but this is a subtle point.*

4. **OCaml Objects** (16 points)

This question also uses the definitions on page 4 of the appendix.

We can encode `Animal` objects as values belonging to an OCaml record type with two fields, `speak` and `setSound`, corresponding to the methods of the same name above.

```
type animal = { speak : unit -> string;
                setSound : string -> unit }
```

Your task in this problem is to implement a "constructor" for such objects—i.e., a function `create` that builds values belonging to this record type. For example, the result of evaluating the following program should be `"duck says QUACK!"`

```
let duck = create "duck" "quack" in
duck.setSound "QUACK!";
duck.speak ()
```

Complete the definition of `create` below to achieve this behavior. Recall that the string concatenation operator in OCaml is written `^`.

```
let create (name : string) (sound : string) : animal =
```

Answer:

```
let sound_ref = ref sound in
{ setSound = (fun (s : string) -> sound_ref.contents <- s);
  speak = (fun () -> name ^ " says " ^ !sound_ref.contents) }
```

Grading Scheme:

- *If a program assumes ocaml's strings are mutable or that the* `speak` *field is mutable, or if a program has serious logic flaws or type errors then points were allocated as follows:*
  - *1 point for returning a record with correct fields*
  - *1 point for the type of the* `speak` *field/function*
  - *1 point for the type of the* `setSound` *field/function*
  - *1 point for having* `speak` *concatenate strings*
  - *1 point for having* `setSound` *mutate some state*
  - *2 points for having mutable state*
  - *1-2 points for logically correct choices that don't fall into the above categories but are in the correct path*
- *If the program was mostly correct, then points were taken off out of 16 as follows:*
  - *-1 for declaring a field mutable when let-initializing a binding*
  - *-2 for declaring a variable mutable*
  - *-4 for assuming a record's field is inherently mutable*
  - *-2 for small type errors*
  - *-1 for not dereferencing*
  - *-1 for declaring a record uninitialized*

- *-1 to -3 for various logic issues*
- *-1 for minor syntax errors if otherwise perfect*
- *-2 for major syntax errors*

5. Java Types and Exceptions (11 points)

Consider the following code, inspired by Homework 9:

```
public abstract class Corrector {
  public abstract Set<String> getCorrections(String wrong);
}

public class SpeelingCorrector extends Corrector {
  public Set<String> getCorrections(String wrong) {
    if (wrong.equals("speeling")) {
      Set<String> results = new HashSet<String>();
      results.add("spelling");
      return results;
    }
    return null;
  }
}
```

a. What is the static type of `results`?
Answer: `Set<String>[]`

b. What is the dynamic class of `results`?
Answer: `HashSet<String>`

c. Can the above code ever produce an exception? If so, which one?
Answer: `NullPointerException`

d. Now consider the following `main` method refering to the definitions above.

```
public static void main(String[] args) {
  Corrector c = new Corrector();
  Set<String> corrections = c.getCorrections("speeling");
}
```

Will this method compile correctly? If not, why not? If yes, what is the value of `corrections` at the end of the execution?

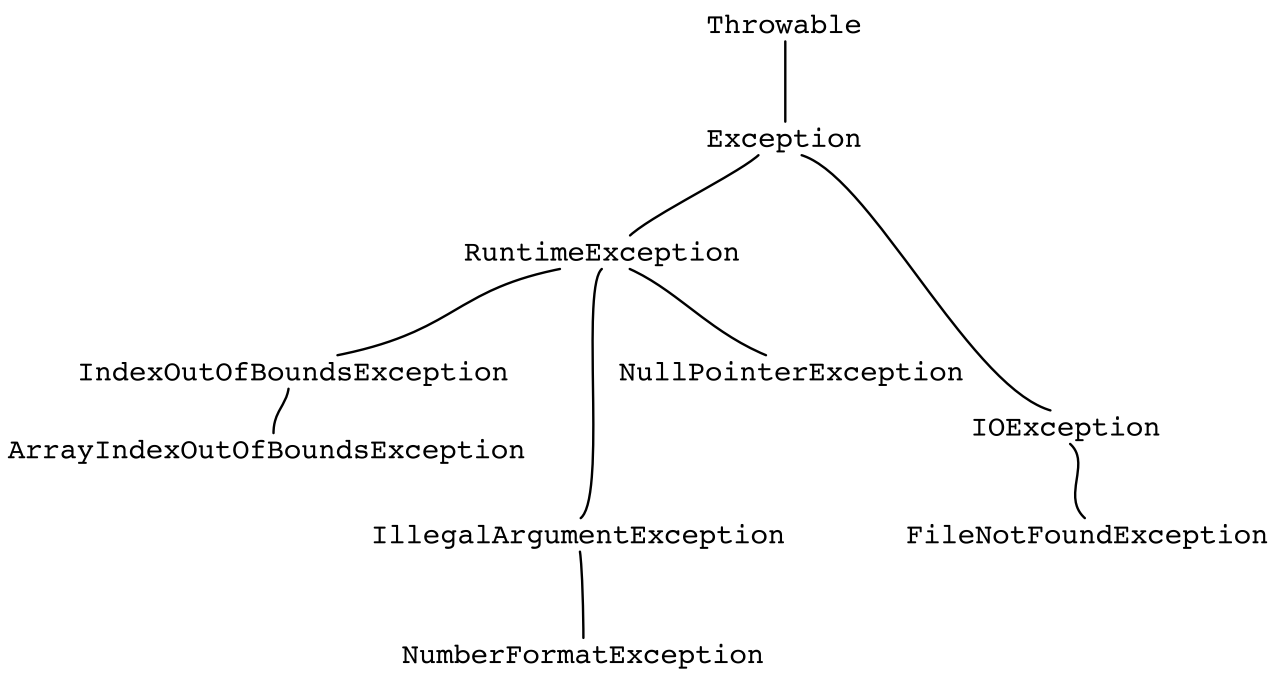
Answer: No: Can't instantiate abstract classes

e. Finally, consider a different `main` method:

```
public static void main(String[] args) {
  Reader r = new FileReader(args[0]);
  char c = (char) r.read();
  SpeelingCorrector s = new SpeelingCorrector();
  Set<String> corrections = s.getCorrections(String.valueOf(c));
}
```

Below, we've drawn a fragment of the exception hierarchy of Java.

Circle the exceptions that can be thrown during the execution of the above piece of code. (Do *not* circle supertypes of the exceptions that can be thrown, unless they are also thrown themselves.) You may find some useful information on page 2 of the appendix.



Answer:

Exceptions that can be thrown:

- `ArrayIndexOutOfBoundsException` can be thrown if the `args` array has zero length
- `FileNotFoundException` can be thrown if `args[0]` is not a valid filename
- `NullPointerException` can be thrown if `main` is called with a `null` argument. (This will never happen if `main` is invoked from outside the program — from the command line or Eclipse — but it can happen if `main` is called from *inside* the program!)
- `IOException` is a subtle case. The javadocs for `Reader` say that the `read` method can throw `IOException`, so it seems reasonable *prima facie* to circle it. However, this declaration by itself does not actually imply that an instance of the `IOException` class *itself*

— as opposed to one of its subclasses — can ever be thrown. To verify that `IOException` itself can indeed be thrown, we need to look carefully at the source code for `FileReader` and its associated library classes. And indeed, there is at least one place, deep in an inner method called `ensureOpen`, where the phrase "**new** `IOException(...)`" literally appears.

Exceptions that *cannot* be thrown:

- `IndexOutOfBoundsException` *can* be thrown by some of the `FileReader` methods, but we believe it cannot be thrown by the code we're dealing with here. Same for `IllegalArgumentException` and `NumberFormatException`.
- `RuntimeException` is not (directly) thrown by any of the code in `FileReader`.
- `Exception` and `Throwable` are declared exceptions; the javadocs for `FileReader` guarantee that they cannot be thrown.

Grading Scheme:

- *1 point each for a and b.*
- *2 points for c, 1 for "yes" and 1 for* `NullPointerException`
- *2 points for d, 1 for "no" and 1 for an explanation involving the phrase (or idea) "abstract class"*
- *5 points for e, allocated by subtracting 1 point for each error (extra circle or missing circle). Exception: circling* `IndexOutOfBoundsException` *instead of* `ArrayIndexOutOfBoundsExc` *costs -1, not -2.*

6. **Collections and Equality** (13 points)

a. The following comment is adapted from the Java library implementation of the `equals` method for both the `LinkedList` and `ArrayList` classes. (The details of how `ArrayList` and `LinkedList` work and how they differ from each other are not important for this problem. Everything you need to know is contained in the italicized text.)

`public boolean equals(Object o)`

Compares the specified object with this list for equality. Returns true if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal. (Two elements `e1` *and* `e2` *are equal if either both are* `null` *or else* `e1.equals(e2)`*.) In other words, two lists are defined to be equal if they contain the same elements in the same order.*

This implementation first checks if the specified object is exactly this list. If so, it returns true; if not, it checks if the specified object is a list. If not, it returns false; if so, it iterates over both lists, comparing corresponding pairs of elements (using `equals`*). If any comparison returns false, this method returns false. If either iterator runs out of elements before the other it returns false (as the lists are of unequal length); otherwise it returns true when the iterations complete.*

Note that both `LinkedList` and `ArrayList` are subtypes of `List`.

Consider the following piece of code that creates some linked lists and arraylists:

```
String str = "CIS 120";

List<String> l1 = new LinkedList<String>();
l1.add(str);

List<String> l2 = new ArrayList<String>();
l2.add(str);

List<String> l3 = l1;
```

For each of the comparisons below, circle whether it returns **`true`** or returns **`false`**.

i. `l1.equals(l2)`
true

ii. `l1 == l2`
false

iii. `l1.equals(l3)`
true

iv. `l1 == l3`
true

v. `l2.equals(l3)`
true

vi. `l2 == l3`
false

b. Consider the following fragment of a `main` method, referring to a `Pair` class. Three possible implementations of `Pair` are given below.

```
// somewhere in main ...
Pair p = new Pair(1,2);
System.out.println(p.equals(new Pair(1,2)));
System.out.println(p.equals((Object) new Pair(1,2)));
```

For each of the following implementations of the `Pair` class, write down what is printed to the console when we call `main`.

i.
```
public class Pair {
  private final int x;
  private final int y;
  public Pair (int x, int y) { this.x = x; this.y = y; }
  public int getX() { return x; }
  public int getY() { return y; }
}
```

Answer:

```
false
false
```

ii.
```
public class Pair {
  // Same declarations and methods as (i)
  ...
  // plus this:
  public boolean equals(Pair that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
}
```

Answer:

```
true
false
```

iii.

```
public class Pair {
  // Same declarations and methods as (i)
  ...
  // plus this:
  @Override
  public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Pair) {
      Pair that = (Pair) o;
      result = (this.getX() == that.getX() &&
                this.getY() == that.getY());
    }
    return result;
  }
}
```

Answer:

```
true
true
```

7. Binary Trees (14 points)

Recall the type definitions for binary trees from homework 2:

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

Also recall that binary search trees (BST) are trees with an additional invariant (which hopefully you remember), and recall the `insert` function for BSTs:

```
let rec insert (x:'a) (t:'a tree) : 'a tree =
  begin match t with
    | Empty -> Node(Empty,x,Empty)
    | Node(lt,y,rt) ->
       if x = y
       then t
       else if x < y
       then Node(insert x lt,y,rt)
       else Node(lt,y, insert x rt)
  end
```

a. Which of the following OCaml values of type `tree` are valid BSTs? (Write "Yes" or "No" by each one.)

i. `let t1 : tree = Empty`

Answer: Yes

ii. `let t2 : tree = Node (Empty, 42, Empty)`

Answer: Yes

iii. `let t3 : tree = Node (insert 42 Empty, 41, Empty)`

Answer: No

iv. `let t4 : tree = insert 42 (Node (Empty, 41, Empty))`

Answer: Yes

v. `let t5 : tree = Node (Node (Empty, 42, Empty), 42, Empty)`

Answer: No

Grading Scheme: 5 pts total. 1 pt each.

b. Also consider the following `insert_list` function using the `fold` higher order function. For your convenience, the definition of `fold` is in the appendix on page 3.

```
let insert_list (l : 'a list) (t : 'a tree) : 'a tree =
  fold insert t l
```

For each of the following code snippets, draw the resulting tree. Here is an example of a drawn tree:

```
      2
     / \
    3   4
       / \
      5   6
```

i. `insert_list [1;2;3;4;5] Empty`

```
          5
         /
        4
       /
      3
     /
    2
   /
  1
```

Grading Scheme: 1pt for correctness.

ii. `Node(insert_list [1;2] Empty, 42, insert_list [42; 120] Empty)`

```
       42
      /  \
     2    120
    /     /
   1     42
```

Grading Scheme: 4 pts total. -1 pt for incorrect fold direction. -2 pts for incorrect leaves but correct root.

iii. `insert_list [4;8;4;3;5;23;42] (insert_list [42] Empty)`

```
        42
       /
     23
     /
    5
   / \
  3   8
   \
    4
```

Grading Scheme: 4 pts total. -1 pt for incorrect fold direction. -2 pts for incorrect tree but has 42 as root. -2 pts for not having 42 as root.

8. Ocaml Programming and Higher Order Functions (16 points)

In this problem you will be guided through the steps of coding a function `replicate`, which repeats all the elements of a list some given number of times.

a. Write a recursive function `repeat` that, given a number `n` and a value `x`, returns a list consisting of the value `x` repeated `n` times. Assume `n` is non-negative.

For example,

```
repeat 3 1 = [1; 1; 1]
repeat 2 'a' = ['a'; 'a']
repeat 0 "cis 120" = []
```

```
let rec repeat (n : int) (a : 'a) : 'a list =
  if n = 0 then [] else a :: repeat (n-1) a
```

b. Write a function `flat_transform` that, given a list `[a1;a2;...;an]` and a function `f`, produces the list `f a1 @ f a2 @ ... @ f an`. For example,

```
flat_transform (fun x -> [x; x+1]) [1;2] = [1;2;2;3]
```

Your solution must use a single call to one of the higher-order functions provided on page 3 of the appendix.

```
let flat_transform (f : 'a -> 'b list) (l : 'a list) : 'b list =
  fold (fun a b -> f a @ b) [] l
```

c. Use functions defined in parts (a) and (b) to write a function `replicate` that takes a non-negative number `n` and a list `l` and returns a new list where each element is repeated `n` times. For example:

```
replicate 2 [1;2;3] = [1;1;2;2;3;3]
```

Hint: You do not need any higher-order functions or recursion here. A simple combination of the functions you have coded above is enough!

```
let replicate (n : int) (l : 'a list) : 'a list =
  flat_transform (repeat n) l
```

9. **Design Process** (8 points)

List the four steps of the “design process” (or “recipe”) that we used throughout the semester.

a. *Understand the problem (and how the concepts relate to each other)*

b. *Define the interface*

c. *Write tests*

d. *Implement (i.e., write the code)*

Grading Scheme: 2 points per step. (This question turned out to be a bit of a waste – nobody got it wrong! – but we hope you enjoyed it. :-)

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.

java.lang

```
public class String
  public String(char[] value)
   // Allocates a new String so that it represents the sequence of
   // characters currently contained in the array argument
  public char charAt(int index)
   // Returns the char value at the specified index
  public int length()
   // Returns the length of this string
  public boolean equals(Object anObject)
   // Compares this string to the specified object. The result is true if and
   // only if the argument is not null and is a String object that represents
   // the same sequence of characters as this object.
   public static String valueOf(char c)
   // Returns the string representation of the char argument.

public class Character
  public static boolean isWhiteSpace(char ch)
     // Determines if the specified character is whitespace
```

java.util (Collections Framework)

```
public interface Iterator<E>
   public boolean hasNext()
      // Returns true if the iteration has more elements. (In other words,
      // returns true if next would return an element rather than throwing an exception.)

   public E next()
      // Returns the next element in the iteration.
      // Throws: NoSuchElementException - iteration has no more elements.
```

java.io

```
public abstract class Reader
  public int 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.
    // 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

public class BufferedReader extends Reader
  public BufferedReader(Reader in)
    // Creates a buffering character-input stream that uses a default-sized input buffer.
    // Parameters: in - A Reader

public class InputStreamReader extends Reader
  public InputStreamReader(InputStream in)
    // Creates an InputStreamReader that uses the default charset.
    // Parameters: in - An InputStream

public class FileReader extends InputStreamReader
  public FileReader(String fileName) throws FileNotFoundException
    // Creates a new FileReader, given the name of the file to read from.
    // Parameters: fileName - the name of the file to read from
    // Throws: FileNotFoundException - if the named file does not exist,
    //         is a directory rather than a regular file, or for some other
    //         reason cannot be opened for reading.
```

Higher order functions

```
let rec transform (f: 'a -> 'b) (x: 'a list): 'b list =
 begin match x with
 | [] -> []
 | h :: t -> (f h) :: (transform f t)
 end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list): 'b =
 begin match l with
 | [] -> base
 | h :: t -> combine h (fold combine base t)
 end

let rec filter (f: 'a -> bool) (l: 'a list) : 'a list =
 begin match l with
   | [] -> []
   | h::t -> if f h then h :: filter f t else filter f t
 end
```

Code for questions 3 - 4

```
public class Animal {
   private String name;
   private String sound;

   public Animal(String name, String sound) {
     this.name = name;
     this.sound = sound;
   }

   public String speak() {
     return (name + " says " + sound);
   }

   public void setSound(String sound) {
     this.sound = sound;
   }
}

public class Duck extends Animal {
   public Duck() {
     super("duck", "quack");
   }
}
```

Main method:

```
public static void main(String[] args) {
  Animal d1 = new Duck();
  Animal d2 = new Animal("duck", "quack");
  /* Here */
}
```

Java ASM

```
public class Counter extends Object {
  private int x;
  public Counter () { super(); this.x = 0; }
  public void incBy(int d) { this.x = this.x + d; }
  public int get() { return this.x; }
}

public class Decr extends Counter {
  private int y;
  public Decr (int initY) { super(); this.y = initY; }
  public void dec() { this.incBy(-this.y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
String s1 = new String("foo");
String s2 = new String("foo");
/* Here */
```

The following picture shows the ASM at the point of the computation marked */* Here */*. Note that we do not show the the `String` class in the class table, and that for the `String` object in the heap we just summarize its contents, ignoring its actual internal representation.