SOLUTIONS

1. **Higher-Order Programming in OCaml** (12 points) Please refer to the OCaml function definitions in Appendix A (most should already be familiar, but you may not recognize `id` or `compose`). Then fill in the blanks to make the following assertions successfully pass, where `assertEq` is defined like this:

```
let assertEq (msg:string) (actual: 'a) (expected: 'a) : unit =
  run_test msg (fun () -> actual = expected)
```

For example, this assertion will pass:

```
assertEq "sample" (transform id [1;2;3;4])
  [1;2;3;4]
```

(a) `assertEq "a" (fold (fun x acc -> 0 :: acc) [] [1;2;3;4])`

```
[0;0;0;0]
```

(b) `assertEq "b" (transform (fun x -> x * 2) (filter (fun x -> x > 2) [1;2;3;4]))`

```
[6;8]
```

(c) `assertEq "c" (transform reverse [[1;2];[3;4]])`

```
[[2;1];[4;3]]
```

(d) `assertEq "d" (compose reverse reverse [1;2;3;4])`

```
[1;2;3;4]
```

(e) `assertEq "e" (reverse (fold (fun x acc -> x :: x :: acc) [] [1;2;3;4]))`

```
[4;4;3;3;2;2;1;1]
```

(f)
```
assertEq "f" (fold (fun x (acc1,acc2) -> (x && acc1, x || acc2))
                   (true,false)
                   [true;true;false;true])
```

```
(false, true)
```

2. **Java Exceptions** (8 points) The code below defines three methods, `m1`, `m2`, and `m3`, that throw and catch exceptions `ExnA` and `ExnB` (two newly declared runtime exceptions that have no relationship with each other). If we start with a call to `m1()`, some of the calls to `System.out.println` will get executed, while others will not. Please mark the appropriate box next to each of these calls to indicate whether the corresponding string will or will not get printed (i.e., put an X inside the ☐ before either `Printed` or `Not printed`).

```
class ExnA extends RuntimeException { }
class ExnB extends RuntimeException { }

static void m1() {
        System.out.println("begin m1");                // ☐ Printed ☐ Not printed
        try {
                System.out.println("calling m2");        // ☐ Printed ☐ Not printed
                m2();
                System.out.println("returned from m2");  // ☐ Printed ☐ Not printed
        } catch (ExnA e) {
                System.out.println("m1 caught ExnA");    // ☐ Printed ☐ Not printed
        } catch (ExnB e) {
                System.out.println("m1 caught ExnB");    // ☐ Printed ☐ Not printed
        }
        System.out.println("end m1");                  // ☐ Printed ☐ Not printed
}

static void m2() {
        System.out.println("begin m2");                // ☐ Printed ☐ Not printed
        try {
                System.out.println("calling m3");        // ☐ Printed ☐ Not printed
                m3();
                System.out.println("returned from m3");  // ☐ Printed ☐ Not printed
        } catch (ExnA e) {
                System.out.println("m2 caught ExnA");    // ☐ Printed ☐ Not printed
                System.out.println("about to throw ExnB"); // ☐ Printed ☐ Not printed
                throw new ExnB();
        } catch (ExnB e) {
                System.out.println("m2 caught ExnB");    // ☐ Printed ☐ Not printed
        }
        System.out.println("end m2");                  // ☐ Printed ☐ Not printed
}

static void m3() {
        System.out.println("begin m3");                // ☐ Printed ☐ Not printed
        try {
                System.out.println("about to throw ExnA"); // ☐ Printed ☐ Not printed
                throw new ExnA();
        } catch (ExnB e) {
                System.out.println("m3 caught ExnB");    // ☐ Printed ☐ Not printed
        }
        System.out.println("end m3");                  // ☐ Printed ☐ Not printed
}
```

Answer:

3. **Java Concepts** (12 points total) Consider the following Java class:

```
public class Student {
    private int    id;
    private String name;
    private String major;

    public Student(int id, String name, String major) {
        this.id = id;
        this.name = name;
        this.major = major;
    }
}
```

(a) (3 points) Will the following test pass?

```
import java.util.LinkedList;

@Test
public void test1() {
     LinkedList<Student> l = new LinkedList<Student>();
     l.add(new Student(1610, "Yakkety Yak", "CSCI"));
     assertTrue(l.contains(new Student(1610, "Yakkety Yak", "CSCI")));
}
```

☐ `Yes` ☒ `No`

If you answered `No`, briefly explain how you would modify the `Student` class (not `test1` itself!) to pass `test1`? (Just explain in words—no need to write the code.)

Answer: Override the `equals` method to compare the components of the object instead of using referential equality.

(b) (3 points) Will the following code compile?

```
import java.util.TreeSet;

@Test
public void test2() {
        TreeSet<Student> s = new TreeSet<Student>();
        s.add(new Student(606, "Dapper Drake", "MSE"));
}
```

☒ `Yes` ☐ `No`

If you answered `No`, how would you modify the `Student` class (again, not the test) to make the code compile?

For the problems on this page, please mark *all* answers that apply.

(c) (2 points) The `repaint` method in Swing is used to ...

☐ perform low-level drawing operations to update the appearance of some portion of the screen
☒ notify the Swing framework that some portion of the screen needs to be updated
☐ quickly update the appearance of some portion of the screen to support real-time animation

(d) (2 points) What is the relation between the *static type* and the *dynamic class* of a Java expression?

☒ The dynamic class will always be a subtype of the static type
☐ The static type will always be a subtype of the dynamic class
☒ An expression only has one static type, but its dynamic class can be different at different points during a program's execution

(e) (2 points) A *cast* expression `(T)e` in Java...

☐ checks that the static type of `e` is exactly `T`
☒ produces a result whose static type is exactly `T` (if it compiles at all)
☒ inserts a runtime check on the dynamic class of `e`

(f) (2 points) In order for a subclass to *override* a method from its superclass, it must...

☐ invoke `super` at the beginning of the overriding method
☐ take arguments that are *subtypes* of the corresponding arguments to the superclass method
☒ take arguments that have *exactly the same* types as the corresponding arguments to the superclass method

(g) (2 points) The term *garbage collection* refers to ...

☐ Refactoring a program to remove unused code
☐ Rebuilding a `Map` data structure to improve its efficiency
☒ Scanning the Java heap to reclaim objects that can no longer be accessed by the program

(h) (2 points) What is a *hash collision*?

☒ The case where two or more keys "hash" to the same "bucket" in a `HashSet` or `HashMap` collection
☐ A runtime error caused by different threads accessing the same element of a `HashMap` at the same time
☐ A bug caused by multiple programmers attempting to modify the same project component

4. **OCaml Objects vs. Java Objects** (26 points total) The standard Java interface `Iterator<A>`

```
interface Iterator<A> {
  A next();
  boolean hasNext();
}
```

corresponds to this OCaml type:

```
type 'a iterator = {
  next : unit -> 'a;
  hasNext : unit -> bool;
}
```

(a) (6 points) Here is an OCaml function `iforall`, which tests whether some boolean predicate `test` returns `true` on every element of a list:

```
let rec iforall (i : 'a iterator) (test : 'a -> bool) =
  if i.hasNext() then
    let x = i.next() in
    test x && iforall i test
  else true
```

To translate this predicate into Java, we first need a way of representing testing functions. For this, we introduce the following interface (a simplified version of the one in Java's `java.util.function` library):

```
interface Predicate<A> {
        boolean test(A arg);
}
```

For example, here is a specific `Predicate` that tests whether its `String` argument is longer than two characters:

```
class LongStringPredicate implements Predicate<String> {
        public boolean test(String x) {
                return (x != null && x.length() > 2);
        };
}
```

(There is nothing to write on this page.)

Fill in the blank in the following Java definition of `iforall`. (The first `<A>` in the header line introduces the generic type parameter `A`; overall, the method header says that, when `iforall` is called, its two parameters, `i` and `pred`, must share the same element type type `A`, just as in the OCaml version.)

Answer:

```
static <A> boolean iforall (Iterator<A> i, Predicate<A> pred) {
        if (i.hasNext()) {
                A x = i.next();
                if (pred.test(x)) return (iforall (i, pred));
                return false;
        } else return true;
}
```

(b) (20 points) See Appendix B for instructions, then fill in the blanks below.

(Note that our solution handles the case where the base iterator can return `null`*, although this was not required. Full credit was given to otherwise-correct solutions that did not handle this case.)*

```
class FilterIterator<A> implements Iterator<A> {
        // Field declarations go here:

        Iterator<A> base;
        Predicate<A> pred;
        A current;
        boolean done;

        private void findNext() {

                done = !base.hasNext();
                if (!done) {
                        current = base.next();
                        if (!pred.test(current))
                                findNext();
                }

        }

        public FilterIterator(Iterator<A> b, Predicate<A> p) {

                base = b;
                pred = p;
                done = false;
                findNext();

        }

        public boolean hasNext() {

                return (!done);

        }

        public A next() {

                if (done) throw new NoSuchElementException();
                A res = current;
                findNext();
                return res;

        }
}
```

5. **Java Programming with Collections** (36 points total) A *bag* (sometimes also called a *multiset*) is an *unordered* collection of values that permits duplicates. Intuitively, a bag is a set that can contain multiple occurrences of the same element. For example, using set-like notation, we might write the bag containing two '1's and one '2' as $\{1, 1, 2\}$ or equivalently as $\{1, 2, 1\}$ or $\{2, 1, 1\}$.

In this problem, you will use the design process to implement (some parts of) a Java `Bag<E>` class.

Step 1 (Understand the problem) There is nothing to write for this step; your answers below will indicate your understanding.

Step 2 (Determine the interface) A `Bag<E>` object implements the `Collection<E>` interface, the relevant parts of which are given in Appendix C.

In addition to the standard collection methods, a bag object provides a method called `getCount(E e)`, which returns a non-negative integer indicating the number of times the element `e` occurs in the multiset.

Step 3 (Write test cases) Below are several example test cases for the `add`, `size`, and `contains` methods of the `Bag<E>` implementation. (No need to do anything but understand them.)

```
@Test
public void sizeEmpty() {
   Bag<Integer> b = new Bag<Integer>();
   assertEquals(0, b.size());
}

@Test
public void size1() {
    Bag<Integer> b = new Bag<Integer>();
    b.add(1);
    assertEquals(1, b.size());
}

@Test
public void size11() {
   Bag<Integer> b = new Bag<Integer>();
   b.add(1);
   b.add(1);
   assertEquals(2, b.size());
}

@Test
public void containsEmpty() {
   Bag<Integer> b = new Bag<Integer>();
   assertFalse(b.contains(1));
}

@Test
public void contains1() {
   Bag<Integer> b = new Bag<Integer>();
   b.add(1);
   assertTrue(b.contains(1));
}
```

(There is nothing to write on this page.)

(a) (5 points) Complete these three distinct tests cases for the `remove` method by filling in the blank with either `True` or `False` indicating whether the assertion should succeed or fail:

```
@Test
public void removeEmpty() {
   Bag<Integer> b = new Bag<Integer>();
   assertFalse(b.remove(1));
}


@Test
public void remove1() {
   Bag<Integer> b = new Bag<Integer>();
   b.add(1);
   assertTrue(b.remove(1));
   assertFalse(b.contains(1));
}

@Test
public void remove11() {
   Bag<Integer> b = new Bag<Integer>();
   b.add(1);
   b.add(1);
   assertTrue(b.remove(1));
   assertTrue(b.contains(1));
}
```

(b) (4 points) Now complete this test case by filling in the blanks so that the test case should succeed. Remember that `b.getCount(e)` should returns a non-negative integer indicating the number of times the element `e` occurs in `b`.

```
@Test
public void countTest() {
   Bag<Integer> b = new Bag<Integer>();
   b.add(1);
   b.add(2);
   assertEquals(1, b.getCount(1));
   assertEquals(1, b.getCount(2));
   b.add(1);
   b.remove(3);
   assertEquals(2, b.getCount(1));
   assertEquals(3, b.size());
}
```

Step 4 (Implementation) To implement the `Bag<E>` class, we must decide how to represent the collection using basic data structures, plus appropriate invariants. Here, we choose to represent the internal state of the `Bag<E>` class using an object that implements `Map<E,Integer>`. The idea is to associate with each element `e` a count of the number of times that `e` occurs in the bag. (Appendix D describes the `Map` interface.)

Let's introduce the notation $[k_1 \mapsto v_1 \ldots k_n \mapsto v_n]$ as shorthand for a `Map<K,V>` object `m` such that `m.get(`k_i`)` returns v_i. For example, we could write

$$[\texttt{"a"} \mapsto 2, \texttt{"b"} \mapsto 1]$$

for the `Map<String,Integer>` object `m` obtained by doing:

```
Map<String,Integer> m = new TreeMap<String,Integer>();
m.put("a", 2);
m.put("b", 1);
```

This object `m` would be a suitable representation of the bag $\{$`"a"`, `"a"`, `"b"`$\}$, with two "`a`"s and one "`b`".

To conveniently implement the `size()` method required by `Collection<E>`, we will also keep track of a `size` field as part of the bag implementation. We thus arrive at this partial implementation the bag class:

```
public class Bag<E> implements Collection<E> {
    private Map<E,Integer> bag;  // representation
    private int size;            // number of elements

    public Bag() {
        bag = new TreeMap<E,Integer>();
        size = 0;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    // continued
```

(There are no questions on this page.)

However, not *every* `bag` object of type `Map<E,Integer>` is a good representation; for example there should never be a negative number of any element in the bag, so we need an invariant to rule out such incorrect maps. Which invariant we choose will affect the difficulty of implementing the methods of the class.

Here are two possible invariants that we might use to rule out invalid maps:

INV1: If `bag.containsKey(e)` then `bag.get(e)` ≥ 0.

INV2: If `bag.containsKey(e)` then `bag.get(e)` > 0.

(c) (2 points) Which of these invariants is assumed by the following implementation of the `contains` operation for `Bag`s?

```
@Override
public boolean contains(Object o) {
    return bag.containsKey(o);
}
```

☐ only INV1 ☒ only INV2 ☐ it works with both INV1 and INV2

(d) (2 points) Which of these invariants is assumed by the following implementation of `getCount`?

```
public int getCount(E e) {
    if (bag.containsKey(e)) {
        return bag.get(e);
    } else {
        return 0;
    }
}
```

☐ only INV1 ☐ only INV2 ☒ it works with both INV1 and INV2

(e) (3 points) Now consider implementing the `equals` method for the `Bag<E>` class: two bags should be considered equal if, for every element `e`, they contain the same number of occurrences of `e`. One of the two invariants **INV1** or **INV2** makes it much simpler to implement the `equals` method. Briefly(!) explain which one and why:

ANSWER: Invariant INV2 makes it easier because we can use the `Map` implementation of `equals`. The presence of "0" values in the map means two different maps with different keys might represent the same bag: e.g. $[$`"a"` $\mapsto 2,$ `"b"` $\mapsto 0]$ and $[$`"a"` $\mapsto 2,$ `"c"` $\mapsto 0]$ both represent the same bag $\{$`"a"`, `"a"`$\}$.

(f) (2 points) Our `Bag` implementation must also maintain an appropriate relationship between the `size` field and the `bag` map. Which of the following invariants correctly expresses that relationship? (Mark one)

☒ If `bag` is the map $[k_1 \mapsto v_1, \ldots, k_n \mapsto v_n]$ then `size` = $v_1 + ... + v_n$.

☐ If `bag` is the map $[k_1 \mapsto v_1, \ldots, k_n \mapsto v_n]$ then `size` = $k_1 + ... + k_n$.

Next, consider the following (buggy but almost correct!) implementation of the `add` method.

```
@Override
public boolean add(E e) {
   if (bag.containsKey(e)) {
       Integer count = bag.get(e);
       bag.put(e, count+1);
   } else {
       bag.put(e,1);
       size++;
   }
   return true;
}
```

(g) (2 points) One of the example test cases that we provided *fails* with this implementation of `add`. Which one? (The other methods are correct and their code is as shown earlier.)

☐ `sizeEmpty` ☐ `size1` ☒ `size11` ☐ `containsEmpty` ☐ `contains1`

(h) (3 points) In one sentence, describe how to fix the bug:

ANSWER: Increment the `size` field in both branches. (Or increment it outside of the conditional statement altogether.)

(i) (13 points) Now implement the `remove` method, the Javadocs for which are:

> **`boolean remove(Object o)`**
>
> Removes a single instance of the specified element from this collection, if it is present. Returns `true` if this collection contained the specified element (or equivalently, if this collection changed as a result of the call) and `false` otherwise.

Use representation invariant **INV2** and the `size` invariant indicated above. Make sure that your implementation passes the test cases. We have reproduced the class declaration and fields here so that you can see them when writing this code. *Hint: pay careful attention to the types, as you will need to use a type cast at some point.*

```
public class Bag<E> implements Collection<E> {
    private Map<E,Integer> bag;  // representation
    private int size;            // number of elements


    @Override
    public boolean remove(Object o) {
        if (bag.containsKey(o)) {
            Integer cnt = bag.get(o);
            cnt = cnt - 1;
            if (cnt > 0) {
                bag.put((E)o, cnt);
            } else {
                bag.remove(o);
            }
            size = size - 1;
            return true;
        } else {
          return false;
        }
    }
```

6. **Java Array Programming** (20 points) Suppose we have a two-dimensional array where each cell is filled with an element of the following enumerated type:

```
public enum Agent {
        A,      // Agent type A
        B,      // Agent type B
        NONE    // Empty cell
}
```

The values `A` and `B` represent two different types of "agents", while `NONE` represents an empty cell.

We would like to know whether the agent in some given cell is *satisfied* with its neighbors, using the following rule:

- An agent is satisfied if at least as many of the cells adjacent to it (horizontally, vertically, or diagonally) are occupied by agents of the same type as are occupied by agents of the other type. That is, if the surrounding cells contain d agents of the other type and s agents of the same type, then this agent is satisfied as long as $s \geq d$.

For example, given this arrangement of agents...

	0	*1*	*2*	*3*
0	NONE	NONE	B	NONE
1	NONE	A	B	A
2	NONE	NONE	**A**	B
3	**A**	NONE	NONE	**A**

... the three agents printed in bold are all satisfied. For example, the `A` agent in the bottom right corner, location $(3, 3)$, is satisfied because it has one neighbor of type `A` and one of type `B`, and the `A` agent at location $(3, 0)$ is also satisfied because it has no neighbors of either type. On the other hand, the `A` agent at location $(1, 1)$ is *not* satisfied because it has two neighbors of type `B` and only one of type `A`, and the `B` agent at location $(0, 2)$ is also not satisfied because it has two neighbors of type `A` and only one of type `B`.

On the next page, please fill in the missing body of the method `satisfied`, which takes an array `agents` and two integer coordinates `row` and `col` and returns `true` if the agent at position `agents[row][col]` is satisfied.

If you need more space (e.g., for a helper method), you can use the blank page at the end of the exam. If you do this, make sure to leave a *clear note* telling us where to look for the extra material.

You may assume that the given coordinates will be within the bounds of the given array, that the given array contains either `A` or `B` (not `NONE`) at the given position, and that the given array is rectangular (every row has the same number of columns) and not empty.

Your solution should correctly handle the "boundary cases" where either `row` or `col` are just barely within the bounds of the array (i.e., exactly on the edge).

(There is nothing to write on this page.)

```java
public enum Agent {
        A,      // Agent type A
        B,      // Agent type B
        NONE    // Empty cell
}


public static boolean satisfied (Agent[][] agents, int row, int col) {
        int height = agents.length;
        int width = agents[0].length;

        int same = 0;
        int diff = 0;
        for (int i = row - 1; i <= row + 1; i++) {
                for (int j = col - 1; j <= col + 1; j++) {
                        if (i >= 0 && i < height && j >= 0 && j < width) {
                                if (i == row && j == col)
                                        continue;
                                if (agents[i][j] == Agent.NONE)
                                        continue;
                                if (agents[i][j] == agents[row][col]) {
                                        same++;
                                } else {
                                        diff++;
                                }
                        }
                }
        }
        return (same >= diff);

}
```

Feel free to use this page as scratch paper. (If you write anything here that you want us to grade, make sure you clearly indicate this on the corresponding page earlier in the exam.)

CIS 120 Final Exam — Appendices

A Higher-Order Functions for Problem 1

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

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

let rec filter (f:'a -> bool) (lst:'a list) : 'a list =
  begin match lst with
    | [] -> []
    | x::xs -> if f x then x :: (filter f xs) else (filter f xs)
  end

let reverse (l: 'a list) : 'a list =
  fold (fun x rest -> rest @ [x]) [] l

let id (x: 'a) : 'a = x

let compose (f: 'b -> 'c) (g: 'a -> 'b) : ('a -> 'c) =
  fun (a: 'a) -> f (g a)
```

B OCaml Iterator Code for Problem 4b

For a more interesting challenge, suppose we want to build a Java class `FilterIterator` that expresses the functionality of the OCaml definition found in Appendix B.

Here is an OCaml definition of an "iterator filter" object:

```
let filterIterator (base: 'a iterator) (test: 'a -> bool) : 'a iterator =
  let current = ref None in
  let rec findNext () =
    if base.hasNext() then
       (let x = base.next() in
        current := Some x;
        if test x then () else findNext())
    else
      (current := None;
       ()) in
  findNext();
  {
    hasNext = (fun () -> (!current <> None));
    next = (fun () -> begin match !current with
                            | None -> assert false
                            | Some(x) -> findNext(); x

                      end)
  }
```

That is, given an iterator `base` and a testing function `test`, the expression

```
filterIterator base test
```

yields an iterator that produces just those elements from `base` for which `test` returns `true`. For example, if `someStrings` is a `string iterator` that produces `"a"`, `"foo"`, `""`, and `"bar"`, then

```
filterIterator someStrings (fun s -> String.length s > 1)
```

is an iterator that produces just `"foo"` and `"bar"`.

Your job will be to translate `filterIterator` into Java.

You may assume that the `base` iterator in the Java version will never return `null`.

Your solution should throw `NoSuchElementException` if the `next` method is called when there are no more elements available.

C Excerpt from the Collections Framework (Lists and Sets) for Problem 5

```java
interface Collection<E> extends Iterable<E> {
  public boolean add(E o);
  // Ensures that this collection contains the specified element
  // Returns true if this collection changed as a result of the call.
  // (Returns false if this collection does not permit duplicates
  // and already contains the specified element.)

  public boolean contains(Object o);
  // Returns true if this collection contains the specified element.

  public int size();
  // Returns the number of elements in this collection.

  public boolean remove(Object o);
  // Removes a single instance of the specified element from this
  // collection, if it is present. Returns true if this collection
  // contained the specified element (or equivalently, if this collection
  // changed as a result of the call) and false otherwise.

  // (Other methods omitted.)
}
```

D Excerpt from the Collections Framework (Maps) for Problem 5

```
interface Map<K,V> {

  public V get(Object key)
  // Returns the value to which the specified key is mapped, or null if this
  // map contains no mapping for the key.
  // More formally, if this map contains a mapping from a key k to a value v
  // such that (key==null ? k==null : key.equals(k)), then this method returns
  // v; otherwise it returns null. (There can be at most one such mapping.)

  public V  put(K key, V value)
  // Associates the specified value with the specified key in this map

  public V remove(Object key)
  // Removes the mapping for a key from this map if it is present.
  // Returns the value to which this map previously associated the
  // key, or null if the map contained no mapping for the key.

  public int   size()
  // Returns the number of key-value mappings in this map.

  public boolean  isEmpty()
  // Returns true if this map contains no key-value mappings.

  public Set<K> keySet()
  // Returns a Set view of the keys contained in this map. The set is backed
  // by the map, so changes to the map are reflected in the set, and
  // vice-versa.

  public boolean  containsKey(Object key)
  // Returns true if this map contains a mapping for the specified key.
}


class TreeMap<K,V> implements Map<K,V> {

  public TreeMap()
  // constructor
  // Constructs a new, empty tree map, using the natural ordering of its keys.

  // ... methods specified by interface

}
```