CSE 331: Software Design & Implementation Winter 2026

Homework 8

Due: Wednesday, March 11th, 11:59pm

The first two problems use the following classes:

```
class Critter {..}
class Marsupial extends Critter {..}
class Blobfish extends Critter {..}
class Quokka extends Marsupial {..}
```

and the following variables:

```
Object obj;
Critter critter;              List<Critter> critters;
Marsupial marsupial;          List<? extends Marsupial> marsupials;
Blobfish blobfish;            List<Blobfish> blobfishes;
Quokka quokka;                List<? super Quokka> quokkas;
```

This is a Quokka (k-woke-a)! A smiley marsupial native to southwestern Australia. It brings you much luck and joy for this homework. Thank you San Diego Zoo :)

Task 1 – QuokkA-round and Find Out [12 pts]

For each line of code below, state whether the call is “true” or “false”. No explanation needed.

a) `marsupials.add(quokka);`

b) `quokkas.add(quokka);`

c) `quokkas.add(marsupial);`

d) `quokkas = critters;`

e) `blobfishes.add(new Blobfish());`

f) `quokkas.add(critter);`

g) `marsupial = marsupials.get(0);`

h) `quokka = marsupials.get(0);`

i) `quokka = quokkas.get(0);`

j) `obj = quokkas.get(0);`

k) `blobfishes.add(null);`

l) `critter = marsupials.get(0);`

Task 2 – A Critter Pill to Swallow [8 pts]

For each claim below, state whether the claim is "true" or "false". No explanation needed.

a) `Collection<Critter>` is a Java subtype of `Set<Critter>`

b) `Set<Marsupial>` is a Java subtype of `Collection<Marsupial>`

c) `Set<Blobfish>` is a Java subtype of `Set<Critter>`

d) `Set<Blobfish>` is a Java subtype of `Collection<Blobfish>`

e) `Critter[]` is a Java subtype of `Blobfish[]`

f) `Set<Critter>` is a Java subtype of `Set<Blobfish>`

g) `Set<Blobfish>` is a Java subtype of `Collection<Critter>`

h) `Blobfish[]` is a Java subtype of `Critter[]`

The next problem uses the following interface from Homework 6. You may assume that the helper math definitions of contains-key(), get-value(), and update-key() are consistent and functional.

```
/**
 * A list of pairs of integers. Each pair (k, v) in the list represents
 * an association between the key k and the value v. As such, the list
 * can be thought of as a map from keys to values.
 */
public interface IntMap {
    /**
     * Returns the number of entries in the map.
     * @return len(obj)
     */
    public int size();

    /**
     * Returns the corresponding value to key k
     * @param k the key whose value is to be retrieved
     * @requires contains-key(obj, k)
     * @return get-value(obj, k)
     */
    public int get(int k);

    /**
     * Adds or replaces the mapping for k so it maps to v.
     * @param k the key to add or update
     * @param v the value for key k
     * @modifies obj
     * @effects obj = (k, v) :: obj_0 if !contains-key(obj, k)
     *          else obj = update-key(obj_0, k, v)
     */
    public void put(int k, int v);

    /**
     * Returns whether the map contains a mapping for key k.
     * @param k the key to check
     * @return contains-key(obj, k)
     */
    public boolean contains(int k);

    /**
     * Removes all entries from the map.
     * @modifies obj
     * @effects obj = nil.
     */
    public void clear();
}
```

Task 3 – CI/CD Typelines [10 pts]

a) Rewrite the IntMap interface to use type parameters so that it can store keys and values that are not just integers. Write the new interface declaration below, naming the interface as Map. Any class implementing Map should be able to sort the keys. You do not need to include any Javadoc comments, just the Java code for the new declaration.

b) Suppose we have the following classes:

```
class Fruit {..}
class Apple extends Fruit implements Comparable<Apple> {..}
class Gala extends Apple {..}
```

Which of the following are legal instantiations of the new interface you wrote above? Briefly explain your choices.

i. `Map<Fruit, String>`
ii. `Map<Apple, String>`
iii. `Map<Gala, String>`

Task 4 – Maxation Without Representation [8 pts]

In this problem, we will further generalize section's `MaxStack` into a `FeatureStack`, replacing the max operation with an arbitrary binary operation $r : (T, T) \to T$. We will use that operation feature to "reduce" a list of values to a certain value by applying r multiple times. For example, the list $1 :: 2 :: 3 :: 4 ::$ nil would be reduced to $r(1, r(2, r(3, 4)))$; for our specific `MaxStack`, this operation r would be the `max` operation, returning the max value of the stack.

Fill in the specification for the `feature` method in the `FeatureStack` interface below, assuming that r is the known binary operation used to reduce. Recall that the abstract state of the stack can be thought of as a list of elements of the established type `T`. As a hint, your return should define a mathematical function reduceAll : List$\langle T\rangle \to T$, which utilizes the given binary operator r. Note that reduceAll does not make sense when the input list is empty, therefore you only need to define it for non-empty lists. You will still need to define a base case and recursive case on the tail of the input list.

```
/**
 * A list of values of type T that acts as a stack. It provides an operation
 * feature that can be specified with a given Reducer.
 */
public interface FeatureStack<T> {
    /**
     * Returns the number of entries in the stack.
     * @return len(obj)
     */
    public int size();

    /**
     * Adds the given value to the top the stack.
     * @param val the entry to add
     * @modifies obj
     * @effects obj = val :: obj_0
     */
    public void push(T val);

    /**
     * Removes and returns the top element of the stack.
     * @requires len(obj) != 0
     * @modifies obj
     * @effects obj_0 = x :: obj
     * @returns x the value removed
     */
    public T pop();

    /** TODO: Your specification goes here ... */
    public T feature();
}
```

Task 5 – Feature's Pet [18 pts]

In this problem, we will finish our work by updating `MaxStackImpl` into `FeatureStackImpl`.
The client using `FeatureStack` will give us this operation r, which is equivalent to a `Reducer<T>` instance in code, as a parameter to the constructor. For this problem, we will assume that the `Reducer` interface is as follows:

```
public interface Reducer<T> {
    T operation(T a, T b);
}
```

If `myReducer` is an instance of this interface, then we could call `myReducer.operation(a, b)` to calculate $r(a, b)$.

With this implmentation, because `this.head` is a `Pairlist`, we want to strip away the values from the pairs to represent the stack. To do so, we define unzip: (PList) → List as such:

$$\text{unzip}(\text{pnil}) := \text{nil}$$
$$\text{unzip}((x, m) :: L) := x :: \text{unzip}(L)$$

```
public class FeatureStackImpl<T> implements FeatureStack<T> {
    // AF: obj = unzip(this.head)
    // RI: <TODO in part (a)>

    private final Reducer<T> reducer;
    private PairList<T> head;
    private int size;

    private static class PairList<T> {
        public final T val;
        public final T reduced;
        public final PairList<T> next;

        public PairList(T val, T reduced, PairList<T> next) {
            this.val = val;
            this.reduced = reduced;
            this.next = next;
        }
    }




    // Continued on the next page...
```

```
        /**
         * Initializes the stack.
         * @param reducer, the Reducer instance to be used for the stack's feature
         * @modifies obj
         * @effects obj = nil
         */
        public FeatureStackImpl(Reducer<T> reducer) {
            // TODO in part (b)
        }

        public int size() {
            return size;
        }

        public void push(T val) {
            // TODO in part (c)
        }

        public T pop() {
            // TODO in part (d)
        }

        public T feature() {
            // TODO in part (e)
        }
    }
```

a) Explain why we need the three fields in `FeatureStackImpl`. Write an RI that uses the `size` field.

b) Fill in the constructor of the class so that it initializes the fields properly, including `reducer`.

c) Fill in the `push` method following proper 331 coding style.

d) Fill in the `pop` method following proper 331 coding style.

e) Fill in the `feature` method following proper 331 coding style.

Task 6 – Extra Credit: A Heap of Faith [0 pts]

In this task, you will write a generic Java interface and specifications for a (mutable) Priority Queue ADT. We will informally describe the ADT and relevant methods below. You must turn these methods into a formal interface. It is up to you to decide the method prototypes, what interfaces to extend from, how to use generics, etc.

A Priority Queue is a data structure that allows you to efficiently store and retrieve values associated with priorities or ranks (rather than the typical FIFO order in a queue). In our case, we will consider a Min Priority Queue, which allows us to read or remove the element of least priority from the priority queue.

Our priority queue should work with any type which accepts a (total) ordering, meaning we should be able to use any type in which we can compare any two instances. We should also be able to iterate over the priority queue, meaning it should share functionality with Java for-each loops. Furthermore, it must contain the following methods:

1. `push` - Add a new element to the priority queue (duplicates are allowed).
2. `peek` - Return, but do not remove the minimum element of the priority queue. The priority queue should not be empty if this is called. Ties may be broken arbitrarily.
3. `pop` - Return and remove the minimum element of the priority queue. The priority queue should not be empty if this is called.
4. `size` - Return the number of elements in the priority queue.
5. Any methods included in any extended interface. You are not required to provide specifications for these methods.

As a reminder, the interface should be generic. You may need to look at documentation or research some relevant Java interfaces to provide some of the above functionality. You may assume any Java libraries you need are already imported.