

# Sample midterm answers for CS 315 - Software Design

1. Consider the following specification for a Counter class and 2 potential subtypes.
  - a. (10 points) For each class, does the implementation meet the specification? If not explain, why not.
  - b. (10 points) For each of the potential subtypes, indicate whether you believe it is a legitimate subtype. In particular, does it follow or violate the rules we discussed concerning how specifications may change in subtypes.

```
public class Counter {
    protected int value;

    /** Initializes the counter to 0 */
    public Counter () {
        value = 0;
    }

    /** Returns the value of the counter */
    public int get() {
        return value;
    }

    /** Increments the counter by 1. */
    public incr () {
        value++;
    }
}

public class Counter2 extends Counter {
    /** Initializes the counter to 0 */
    public Counter2 () {
        value = 0;
    }

    /** Doubles the counter. */
    public incr () {
        value = value * 2;
    }
}

public class Counter3 extends Counter {
    /** Initializes the counter to 0 */
    public Counter3 () {
        value = 0;
    }
}
```

```

/** Initializes the counter to n */
public Counter3 (int n) {
    value = n;
}

/** Increases the value in the counter by n.
 * @param n the value to increase by. n must be > 0.
 */
public incr (int n) {
    if (n > 0) {
        value = value + n;
    }
    else {
        value = n;
    }
}

```

Part a:

The implementation of all 3 classes meet their specifications. The only thing at all tricky is in the incr method of Counter3. It has a precondition and the method does as it should if the precondition holds. If the precondition doesn't hold, the method is allowed to do anything and, in fact, does something kind of bizarre.

Part b:

Counter2 is not a valid subtype. The incr method does not have the same or stronger postcondition.

Counter3 is a valid subtype. It only adds behavior.

2. For this question, I would like you to examine the API for part of the Collections hierarchy provided by Java. The overview comment for `java.util.Collection` states:

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The SDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

*Bags* or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this

convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries comply.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the collection. For example, invoking the `addAll(Collection)` method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Reading on we see that `Collection` specifies the following methods as required of all implementations:

- `public int size ()`
- `public boolean isEmpty ()`
- `public Iterator iterator ()`
- `public Object [] toArray ()`
- `public Object [] toArray (Object [] a)`
- `public boolean equals (Object o)`
- `public int hashCode ()`

The following methods may fail depending on what types of things the collection contains, for example, they may throw `ClassCastException` if the collection contains only instances of a specific type (like `Polynomial`) and the parameter passed in does not have that type:

- `public boolean contains (Object o)`
- `public boolean containsAll (Collection c)`

The following methods may be unsupported in any implementing class, that is, their implementations may throw `UnsupportedOperationException`:

- `public boolean add (Object o)`
- `public boolean addAll (Collection c)`
- `public boolean remove (Object o)`
- `public boolean removeAll (Collection c)`
- `public boolean retainAll (Collection c)`
- `public void clear ()`

- a. (10 points) Examine the documentation for the `Collection` interface. You are free to look at the online API documentation provided by Sun. If

somebody writes the following code, what possible outcomes are there? You may assume the code compiles.

```
public void addToCollection (Collection c, Object o) {  
    c.add (o);  
}
```

- b. (15 points) Propose another way to organize the Collections hierarchy to provide mutable collections and immutable collections. Your reorganization should provide compile-time errors if the programmer attempts to modify an immutable collection.
- c. (15 points) Why do you think Sun organized their Collection hierarchy the way they did rather than the way you proposed in part c?
  - a. **NullPointerException** if c is null.  
**NullPointerException** if o is null.  
**Object added and true is returned.**  
**Object already in collection, false is returned.**  
**UnsupportedOperationException** if the actual object in c does not support add.  
**ClassCastException** if the collection only allows objects of a particular class and o is not of that class  
**IllegalArgumentException** if the object is not in the collection already and the collection does not add the object for any reason than those given above. For example, maybe the collection is only intended to hold positive Integers and a negative value was passed in.
  - b. **Collection should have 2 subclasses: ImmutableCollection and MutableCollection.** **MutableCollection** adds mutator methods like add, remove and clear. **ImmutableCollection** does not add any operators that allow mutation. For every kind of collection currently in the collections hierarchy, like Set, LinkedList, etc., there would now be two versions. One version is immutable; the other is mutable. Since add is not defined on the immutable classes, any attempt to call add would result in a compile-time error rather than the run-time error of the current organization.
  - c. One big problem is that immutability is not enforceable, short of making a class final. We can't make **ImmutableCollection** final because then we couldn't have things like **ImmutableSet**, **ImmutableList**, etc. Because of this, there is no way to prevent someone from writing a subclass that is mutable but claiming it extends **ImmutableCollection**. A programmer declaring an object to be an **ImmutableCollection** but ending up with a mutable object in the variable would be surprised (or worse!).

Since **MutableSet** is a lot like **ImmutableSet** except that it has additional methods. So, it would be very tempting to make **MutableSet** as subclass of **ImmutableSet** so that it could inherit the common code. This would lead to the problem that a mutable set could masquerade as an immutable set.

The main problem with these solutions is that if a mutable set masqueraded as an immutable set, the problem would not be detected by the compiler or runtime system. With Sun's solutions if we try to mutate an immutable collection, we will at least get a runtime error. This is better than no error at all, which our proposed alternative is susceptible to.