CSE 331 22wi Final Exam 3/15/22 Sample Solution

Question 1. (12 points) Comparing specifications. Here are four different specifications for a method that searches an integer array `a` to find the location of a negative number in the array `a`.

Specification A
@requires a != null and a.length > 0 and at least one element of a is negative (< 0)
@return k such that a[k] < 0

Specification B
@requires a != null and a.length > 0 and at least one element of a is negative (< 0)
@return k such that a[k] < 0 and all elements in a[0..k-1] are >= 0

Specification C
@requires a != null and a.length > 0 and exactly one element of a is negative (< 0)
@return k such that a[k] < 0

Specification D
@requires a != null and a.length > 0
@return k such that a[k] < 0
@throws NoSuchElementException if no element of a is negative (< 0)

Now suppose we have four different implementations I1 through I4, each of which is known to satisfy one of the above specifications. Since an implementation that satisfies a stronger specification will also satisfy a weaker specification, it's entirely possible that some of the implementations might satisfy additional specifications beyond the one already shown below in the table.

Add an X in the following table in every square where the implementation given in the left column will also satisfy the specification shown in the top row. An X has already been supplied for each implementation and the specification it is known to satisfy.

impl \ spec	spec. A	spec. B	spec. C	spec. D
impl. I1	X		**X**	
impl. I2	**X**	X	**X**	
impl. I3			X	
impl. I4	**X**		**X**	X

It's mid-March and yesterday was π-day (3/14). Being CSE 331-trained software specialists, it seems like it would be a good idea to create some code to keep track of various fruits that could be used as pie fillings. Here is the code for several related classes. Please **leave this page in the exam.** An extra copy of this page is included at the end of the exam that you can remove for convenience while working. Answer questions about this code on the next few pages.

```
/** Fruit for Pie fillings */
class Fruit {
  /** cut this Fruit into n slices */
  void slice(int n) { System.out.println(n + " Fruit slices"); }
}

class Cherry extends Fruit {
  /** cut this cherry into two slices */
  void slice() { slice(2); System.out.println("Cherry"); }
}

class Apple extends Fruit {
  /** return the taste of this apple */
  String flavor() { return "yummy"; }
  /** cut this apple into 8 slices */
  void slice() { slice(8); System.out.println("Apple"); }
}

class Fuji extends Apple {
  /** return the taste of this apple */
  String flavor() { return "sweet"; }
  /** cut this apple into n slices */
  void slice(int n) { System.out.println(n + " Fuji slices"); }
}

class GrannySmith extends Apple {
  /** return the taste of this apple */
  String flavor() { return "tart"; }
  /** cut this apple into 12 slices */
  void slice() { slice(12); System.out.println("Granny slices");}
}
```

Do not remove this page from the exam, but feel free to tear off the copy at the end of the exam. Continue with questions about this code on the next page.

Question 2. (15 points). Overloading and overriding. The following table contains in the left column lines of code from a main program that uses the fruit classes on the previous page. Your job is to write in the right column the output produced when the corresponding line of code is executed, or, if there is something wrong with that line of code that keeps it from executing successfully such as a compile-time or run-time error, you should give a very brief explanation of the problem (like "compile error – no such method in ChocolateCake"). If a line of code produces more than one line of output, write all of the output in the table entry in the correct order. If a line of code does not have any errors and produces no output, leave the corresponding entry in the table blank. The lines of code are executed in order (or at least attempted in the given order, although some may not execute due to errors)

a) `Fruit c = new Cherry();`	
b) `c.slice(10);`	**`10 Fruit slices`**
c) `Fruit yum = new Apple();`	
d) `System.out.println(yum.flavor());`	**Error – no `flavor()` method in `Fruit`**
e) `yum.slice();`	**Error – no `slice()` method in `Fruit`**
f) `Apple a = new Apple();`	
g) `System.out.println(a.flavor());`	**`yummy`**
h) `a.slice();`	**`8 Fruit slices \n Apple`**
i) `a.slice(3);`	**`3 Fruit slices`**
j) `Apple f = new Fuji();`	
k) `f.slice();`	**`8 Fuji slices \n Apple`**
l) `f.slice(3);`	**`3 Fuji slices`**
m) `GrannySmith gs = new GrannySmith();`	
n) `System.out.println(gs.flavor());`	**`tart`**
o) `gs.slice();`	**`12 Fruit Slices \n Granny slices`**

Note: \n is used above to show breaks between multiple output lines written when the corresponding code is executed. Answers did not need to include this – as long as the correct output was given, an answer received full credit.

Question 3. (12 points, 1 each) And now for the dreaded generics question. ☺ Using the `Fruit` class hierarchy from the previous pages, assume we have the following variables:

```
Object obj = null; Fruit fr = null; Cherry c = null;
Apple a = null; Fuji f = null; GrannySmith s = null;

List<? extends Apple> exta = new ArrayList<Apple>();
List<? extends Fuji> extf = new ArrayList<Fuji>();
List<? super Apple> supa = new ArrayList<Apple>();
```

For each of the following, circle OK if the statement has correct Java types and will compile without type-checking errors; circle ERROR if there is some sort of type error. (Note that question only asks about type checking, so it doesn't matter whether the argument 1 in get(1) is out-of-bounds or not. The type checker also does not consider the actual values stored in variables when deciding if they are being used properly.)

OK (ERROR) `exta.add(a);`

OK (ERROR) `exta.add(f);`

(OK) ERROR `exta.add(null);`

OK (ERROR) `extf.add(f);`

(OK) ERROR `supa.add(s);`

OK (ERROR) `supa.add(obj);`

(OK) ERROR `obj = extf.get(1);`

(OK) ERROR `a = exta.get(1);`

OK (ERROR) `f = exta.get(1);`

(OK) ERROR `a = extf.get(1);`

OK (ERROR) `a = supa.get(1);`

(OK) ERROR `obj = supa.get(1);`

Specifications and generic things. Now that we have classes to represent fruits of various sorts, it seems like we should implement a class to hold a basket of fruits. One of the new interns hacked up this class in an hour and it seems like a decent start, but, in the usual CSE 331 exam style, it is lacking in various forms of documentation and might have some problems – or maybe it does work correctly.

Answer questions about this code below and on the next few pages. Please **leave this page in the exam.** An extra copy of this page is included at the end of the exam that you can remove for convenience while working. ☞ Don't forget the question at the bottom of this page! ☜

```
// cse331 22wi final exam - fruit basket
public class Basket {
  private final List<Fruit> items; // items in this basket

  public Basket(){
    items = new ArrayList<Fruit>();
  }

  public void add(Fruit f) {
    items.add(f);
  }

  public List<Fruit> getItems() {
    return items;
  }

  public int getSize() {
    return items.size();
  }
}
```

Do not remove this page from the exam, but feel free to tear off the copy at the end of the exam. Continue with questions about this code below and on the next pages.

Question 4. (2 points) This `Basket` class is basically a wrapper around a `List` instance variable. Which design pattern is illustrated by the overall organization of this class? (Circle the correct answer)

Factory	Singleton	Prototype	Builder
(Adaptor)	Composite	Decorator	Proxy
Iterator	Observer	Strategy	Visitor

This is a use of a wrapper pattern for a `List`. Since the `Basket` interface is different from the underlying `List` interface, it is an Adaptor, not a Decorator or Proxy.

Question 5. (10 points) Class specification. This class is lacking the appropriate CSE331-style documentation. For this question, supply a proper abstract description of the class, a rep invariant, and an abstraction function. You should base your specifications on the intended behavior inferred from the original code and comments.

(a) (3 points) Give an appropriate abstract description of the class that should appear in the JavaDoc comment right before the first line of the class.

```
/**
 * A Basket is a mutable unordered collection of Fruit
 * objects { f1, f2, ..., fn } that may contain duplicates.
 * Items in the Basket are not null.
 *
 */
public class Basket { ... }
```

Grading note: The existing code will work even if `null` values can be added to a `Basket`, but then the specification for `getSize()`, at least, is somewhat unnatural, since normally that would be expected to return the number of `Fruit` objects in the `Basket`. The easiest way to resolve this is to simply disallow `nulls` in a `Basket`, as done here. However, if `nulls` are allowed, there was no penalty as long as the other specifications for methods, the rep invariant, and the abstraction function were consistent with that choice.

(b) (4 points) Give a suitable representation invariant for this class. You should use the existing instance variable that is already present in the code (i.e., you should use the rep that is already there). You should make any appropriate assumptions about how the instance variable is used, but should not add more constraints to the rep invariant than are needed for correct functioning.

`items != null` and `items` does not contain `null` values

(c) (3 points) Give a suitable abstraction function for this class. Your answer should use information from your answers to parts (a) and (b) of the question as needed.

The entries in `items[0..items.size()-1]` represent the `Fruit` objects `{ f1, …, fn }` that make up the contents of this `Basket`.

Question 6. (10 points, 5 each) Method specifications. Give appropriate CSE331-style JavaDoc specifications for methods `add` and `getSize`. For CSE331 custom tags like @requires (or any others) you are free to write @requires or @spec.requires. You should base your specifications on the intended behavior inferred from the original code and comments.

```
/**
 *  Add a new piece of Fruit to this Basket
 *
 *  @param f Fruit object to add to this
 *
 *  @requires f != null
 *
 *  @modifies this
 *
 *  @effects Fruit object f added to this
 *
 *
 *
 *
 */
public void add(Fruit f) { ... }

/**
 *
 * Return the number of Fruit objects in this Basket
 *
 * @return number of Fruit objects contained in this
 *
 *
 *
 *
 *
 */
public int getSize() { ... }
```

Note: The synopsis line (the first line of text in each JavaDoc comment before the tags like `@param`) absolutely is required in a correct JavaDoc specificaiton, otherwise the method does not have a proper 1-line description in the class summary in the generated JavaDoc web pages. In some old exams the synopsis line was omitted by accident or supplied in the starter code given with the problem, so many people missed this. We decided not to deduct points if it was omitted, but only because of possible inconsistencies with previous published solutions, not because it is not needed.

Question 7. (5 points) Are there any representation exposure problems in the `Basket` class? (circle)

(Yes) No

If there are representation exposure problems, give a brief description of what's wrong and describe one way to fix the problem. If there are no representation exposure problems, leave the rest of this question blank.

Method `getItems()` returns a reference to the `items` instance variable, which would allow clients to directly modify the representation data without accessing the `Basket` through appropriate public operations.

The easiest solution is to have `getItems()` return a duplicate or unmodifiable copy of the `items` list.

Note: we assume that `Fruit` objects are immutable and do not need to be copied, since they contain no state that can be modified. Answers that said it could be necessary to do a deep copy that would also copy all of the `Fruit` objects also received full credit if they made the assumption that Fruit objects were mutable.

Question 8. (3 points) We would like to add an overloaded add method to this class that clients could use to add all elements from any suitable collection to the contents of this fruit Basket. The method would look like this:

```
public void add( Collection<? extends Fruit> c) {
  items.addAll(c);
}
```

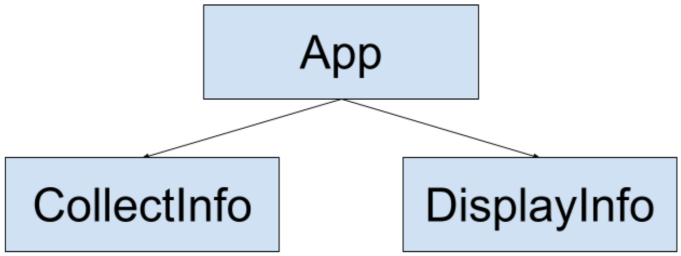
What would be the best parameter type to write in the blank space in the above method's parameter list to allow this method to accept any Java collection as an argument provided that the collection elements can be added to this `Fruit Basket`? (write your answer in the blank space provided in the method parameter list above)

Question 9. (8 points) Generic classes. The code in the original `Basket` class actually is quite general and does not have any real dependencies on particular properties of the `Fruit` objects that are stored in it. Here is the original code for `Basket` again. Show the modifications needed to change this class to a generic class where the elements in the `Basket` are specified by a type parameter such as `Basket<Fruit>` or `Basket<String>`. You should show the needed changes by writing in new additions to the code or crossing out existing code and replacing it with updated code to add the generic element type to the class. You should not make any other changes to the code or add any other methods to it (specifically, don't include the new `add` method that calls `addAll` that was the subject of the previous question, and don't repair any representation exposure problems if they exist – just add generics to the existing initial code below).

Changes needed shown in bold green. We need to add a generic type parameter `<E>` to the class and then replace all occurrences of type `Fruit` with `E` throughout the code.

```
public class Basket<E> {
  private final List<E> items; // items in this basket

  public Basket(){
    items = new ArrayList<E>();
  }

  public void add(E f) {
    items.add(f);
  }

  public List<E> getItems() {
    return items;
  }

  public int getSize() {
    return items.size();
  }
}
```

Question 10. (12 points) React. Consider the following React application structure:



This application collects the user's first name (string), last name (string), eye color (string), and age (number), and displays the information to the user before they can submit it. The three components have the following behavior:

`App:` The top component. Should store and pass down user information. Additionally, it includes four functions with the following stubs:

```
setUserFirstName(a: string)
setUserLastName(a: string)
setUserEyeColor(a: string)
setUserAge(a: number)
```

These functions can be used to set the fields in `App`'s state.

`CollectInfo:` Includes an interactive form. As the user enters their information, this component should update `App` using the callback functions defined above after performing the proper input validation and error handling.

`DisplayInfo:` This component should receive user information from `App` and display it.

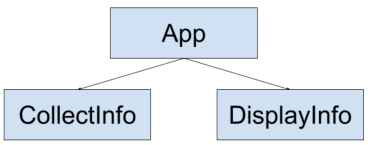
(problem continued on the next page)

Question 10. (cont.) Our React code also defines these interfaces:

```
interface One {
   first: string;
   last: string;
   eyeColor: string;
   age: number | undefined;
}

interface Two {
   first: string;
   last: string;
   eyeColor: string;
   age: string;
}

interface Three {
   name: string;
   eyeColor: boolean;
   age: string;
}

interface Four {
   first: string;
   last: string;
   eyeColor: string;
   age: number | undefined;
   isDisplayed: boolean;
}

interface Five {
   onFirstNameChange: (a: any)=>void;
   onLastNameChange: (a: any)=>void;
   onEyeColorChange: (a: any)=>void;
   onAgeChange: (a: any)=>void;
}

interface Six {
   onFirstNameChange: (a: any)=>any;
   onLastNameChange: (a: any)=>any;
   onEyeColorChange: (a: any)=>any;
   onAgeChange: (a: any)=>any;
}
```

Below are the headings for the three classes `App`, `CollectInfo`, and `DisplayInfo`. Your job is to fill in the blanks with the correct interfaces from the above list so that the components will work properly. An interface might be used more than once or might not be needed at all. If more than one interface could be used in a particular space, pick the **more restrictive** one. If no interface is needed in a particular space, that should be indicated as usual using an empty pair of braces `{}`.

```
class App extends Component< { } , One > {…
class CollectInfo extends Component< Five , Two > {…
class DisplayInfo extends Component< One , { } > {…
```



Grading note: It turns out that `One` and `Two` are both fairly reasonable answers to use as state for `CollectInfo`. Since the age is represented by a number, at some point you will need to convert the user's input string into a numeric value. The conversion will likely happen in `CollectInfo` if you choose `One` and in `App` if you choose `Two`. In other words, this design choice boils down to deciding which component should be responsible for converting the validated input into a number.

Question 11. (6 points, 2 each) Creational patterns. Suppose we have a program that handles orders for a pizza restaurant. The program includes a class `Pizza` with many, many subclasses for different kinds of pizzas, like `VeggiePizza`, `PepperoniPizza`, and so forth. In different parts of the program there are various places where we need to create new `Pizza` objects. Here are three examples of code that create a new `Pizza` of some sort using one of the creational design patterns. Circle the name of the pattern that is the best match to the code sequence:

(a)
```
Pizza pie = genericPizza.clone();
```

Factory Singleton (Prototype) Builder Interning Dependency Injection

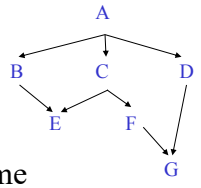
(b)
```
Pizza pie = new PizzaMaker()
                    .cheese("mozzarella")
                    .toppings("mushrooms")
                    .toppings("spinach")
                    .size(size.LARGE)
                    .make();
```

Factory Singleton Prototype (Builder) Interning Dependency Injection

(c)
```
Pizza pie = getNewPizza();
```

(Factory) Singleton Prototype Builder Interning Dependency Injection

Question 12. (3 points) System integration. When building a large system, there are two common strategies for the order in which to implement, combine, and test the different parts of the system: top-down and bottom-up. These two strategies have different characteristics and strengths. Assuming a project is an ordinary one, not relying on unexpected technological innovations or other unusual breakthrough, what is often the best overall strategy for building and integrating the parts: top-down, bottom-up, or some combination of the two? Give a brief justification for your answer.



A combination is usually a good overall strategy. Primarily top-down is best to make progress visible to the team and clients early, and to reveal significant design flaws sooner when it is easier and less costly to fix them. However, selective bottom-up implementation is important to include to be sure that major infrastructure pieces are feasible and will be able to provide the performance needed.

Question 13. (2 free points) (All reasonable answers receive the point. All answers are reasonable as long as there is an answer. ☺)

Draw a picture of something that you plan to do during spring break!

Congratulations from the CSE 331 staff!
Have a great break and see you in the spring!!