CSE 331 Spring 2018 Final

Name ____

There are 8 questions worth a total of 100 points. **Please budget your time so that you get as many points as possible.** We have done our best to make a test that folks can complete in 110 minutes, but everyone works at a different pace, and that is just fine!

The exam is closed book, closed electronics, closed classmates, open mind. Many of the questions have short answers, even if the prompt is a little long. Don't worry!

For all questions involving proofs, assertions, invariants, etc., please assume that all integer quantities are unbounded (e.g., overflow cannot happen) and that ***integer division and square root (sqrt) are truncating as in Java***, i.e., 5/3 evaluates to 1 and sqrt(17) evaluates to 4.

If you do not remember the syntax of some command or the format of a command's output, make the best attempt you can. We will not be grading syntactic details.

Relax and have fun! We're all here to learn.

Please wait to turn the page until everyone is told to begin.

1. ________________ / 10	5. ________________ / 20
2. ________________ / 12	6. ________________ / 20
3. ________________ / 8	7. ________________ / 12
4. ________________ / 12	8. ________________ / 6

QUESTION 1: Warm Up (10 points)

For each statement below, please indicate whether it is true or false by clearly circling the appropriate answer on the right side.

In general, code with a stronger specification is harder to implement but easier to use.	True	/	False
Methods should ***always*** checks their preconditions (@requires) to ensure arguments are valid.	True	/	False
Java subtypes are always true subtypes.	True	/	False
If A is a true subtype of B, then an instance of A can be safely substituted in for an of B.	True	/	False
The Java type system ensures that objects only refer to fields and call methods that are defined.	True	/	False
The Java type system ensures that all exceptions are either caught or declared in a method's throws clause.	True	/	False
Array subtyping in Java is covariant.	True	/	False
Debugging is easier than writing code, so it is OK to be as clever as possible when developing your program.	True	/	False
Sometimes you should call `paintComponent()` in Swing. False	True	/	
Sometimes you should call `repaint()` in Swing.	True	/	False

QUESTION 2: Generics and Wildcard Bounds (12 points)

Given this class hierarchy:

```
class Pet
class Cat extends Pet
class Dog extends Pet
class Husky extends Dog implements Mascot
```

and the following variables:

```
Object o; Pet p; Cat c; Dog d; Husky h; Mascot m;
List <? extends Pet> lep;
List <? extends Cat> lec;
List <? super Husky> lsh;
```

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.

OK ERROR `lsh.add(h);`

OK ERROR `lep.add(o);`

OK ERROR `lep.add(d);`

OK ERROR `lec.add(p);`

OK ERROR `lsh.add(o);`

OK ERROR `lec.add(null);`

OK ERROR `d = lsh.get(1);`

OK ERROR `d = led.get(1);`

OK ERROR `d = lep.get(1);`

OK ERROR `o = led.get(1);`

OK ERROR `p = lsh.get(1);`

OK ERROR `p = null.get(1);`

QUESTION 3: Subtyping and Generics (8 points)

Given this class hierarchy (repeated from the previous problem):

```
class Pet
class Cat extends Pet
class Dog extends Pet
class Husky extends Dog implements Mascot
```

For each statement below, please indicate whether it is true or false by clearly circling the appropriate answer on the right side. Remember that Set implements Collection.

Set<Pet> is a Java subtype of Collection<Pet>	True	/	False
Collection<Pet> is a Java subtype of Set<Pet>	True	/	False
Set<Cat> is a Java subtype of Set<Pet>	True	/	False
Set<Pet> is a Java subtype of Set<Cat>	True	/	False
Set<Cat> is a Java subtype of Collection<Pet>	True	/	False
Dog[] is a Java subtype of Pet[]	True	/	False
Pet[] is a Java subtype of Dog[]	True	/	False
Subtyping can be confusing	True	/	False

QUESTION 4: Specification Strength and Substitutability (12 points)

Here are 4 specifications for a `search()` method that finds paths between nodes in a graph. All specifications have the same "`@param`" Javadoc entries:

```
@param graph - graph to search
@param source - node to start search from
@param sink - node to end search to

S1:   @requires graph contains at least one path from source to sink
      @return a path from source to sink in graph

S2:   @requires graph contains at least one path from source to sink
      @return the shortest path from source to sink in graph

S3:   @requires graph != null
      @return a path from source to sink in graph if one exists,
              otherwise null

S4:   @requires graph contains no cycles
      @return the shortest path from source to sink in graph if one exists,
              otherwise null
```

We have four different implementations of the `search()` method, A, B, C, and D. Each of these implementations is known to satisfy at least one specification, as shown in the table below (i.e., A satisfies S1, B satisfies S2, C satisfies S3, and D satisfies S4).

Your job is to add additional X's in the table for the cases where we can conclude that an implementation satisfies additional specifications given that it is known to satisfy the specification already given in the table. (i.e., given the specifications above, and the known *"implementation xi satisfies Si"* information, which other specifications are also satisfied by each of the implementations?)

	S1	**S2**	**S3**	**S4**
A	**X**			
B		**X**		
C			**X**	
D				**X**

QUESTION 5: Event-driven Programming and Module Dependencies (20 points)

The `eventLoop()` method in the `Main` class of the provided code (on a separate sheet) is fairly straightforward: it loops forever and on each iteration gets a `Message m` and handles it by calling `handle(m)` on the appropriate `Handler` based on the sender (represented as an integer id).

However, `eventLoop()` also has several design decisions fixed in the code that could be hard to modify: handlers cannot be dynamically added or removed from listening for messages, the polling technique (waiting for messages) is "baked in" which can make it tricky to change how the code listens for new messages, and having a single handler respond to messages from different senders requires copy/pasting code between cases.

(a) To address these issues, your buddy Susan developed more general `Dispatcher` and `Listener` classes shown on the previous page. Rewrite `eventLoop()` to use these methods and avoid any explicit loops or conditionals/switches in method. *(Hint: Our solution is ~ 10 lines.)*

```
void eventLoop() {










}
```

(QUESTION 5 continued)

Remember that Module Dependency Diagrams (MDDs) are graphs where an edge is drawn from module A to module B if module A depends on module B.

(b) Draw the edges in the MDD for the code with the original version of `eventLoop()`

`Message` `Handler` `Main`

`Dispatcher` `Listener`

(c) Draw the edges in the MDD for the code with your updated version of `eventLoop()`

`Message` `Handler` `Main`

`Dispatcher` `Listener`

(d) Are there fewer or more edges in the updated MDD? What does this imply about the relative complexity of the two strategies? Which will be easier to extend and why?

QUESTION 6: Testing and Debugging (20 points)

Effectively testing and debugging event-driven code can be challenging; just think of your most subtle challenge from Campus Maps in HW9!   In this question we will think about testing and debugging for the `Dispatcher` class from the code used for Question 5.

(a) Write a brief specification of how the `register()` and `dispatch()` methods should work together.

(b) Describe three black-box tests that ensure the `register()` and `dispatch()` methods work together correctly. If there is a bug in these methods, describe at least one test that reveals the defect and propose a fix.

(QUESTION 6 continued)

(c) Expand your specification from part (a) to now describe how the `register()`, `unregister()`, and `dispatch()` methods should work together.

(d) Describe three black-box tests that ensure the `register()`, `unregister()`, and `dispatch()` methods work together correctly. If there is a bug in these methods, describe at least one test that reveals the defect and propose a fix.

QUESTION 7: Short Answer (12 points)

Please keep your responses short and clear. A single sentence will usually suffice.

(a) Describe a situation that demonstrates why an object that overrides `equals` should also override `hashCode`.

(b) What limitation of Java constructors inspired the Factory design pattern (and many other creational patterns)?

(c) Could it be appropriate to use the Interning design pattern with `RatNum` objects from Homework 4? If so, describe a situation when it might be useful. If not, justify why not.

(d) Could it be appropriate to use the Interning design pattern with a graph ADT like the from Homework 5? If so, describe a situation when it might be useful. If not, justify why not.

(e) Please rank the following strategies for dealing with bugs from best (1) to worst (4):

_____ Make errors immediately visible

_____ Debugging

_____ Use tools that make errors impossible

_____ Write code carefully to avoid introducing bugs

QUESTION 8: Reflecting on the Bigger Picture (6 points)

Please choose ***one*** of the following prompts and write a brief paragraph on that topic below:

- How could applying the ideas you learned in 331 help to make the world a better place?
- What 331 topic do you wish you had learned earlier? When would it have helped?
- What 331 topic do you think will be most useful in the future? Why?

THANK YOU FOR A GREAT QUARTER :)

HAVE A SPECTACULAR SUMMER AND GOOD LUCK IN EVERYTHING!

Questions 5 and 6 refer to this code which sketches a simple event loop that dispatches messages received from some source to the appropriate handler.

```
/* BASIC VERSION */

class Message {
  int sender() {
     ... /* return sender id */
  }
}

class Handler {
  Handler(int id) {
    ... /* initialize handler for Messages from sender id */
  }
  void handle(Message m) {
    ... /* handle a Message */
  }
}

class Main {
  void eventLoop() {
    // make handlers for messages from various senders
    Handler h1 = new Handler(1);
    Handler h2 = new Handler(2);
    Handler h3 = new Handler(3);

    // forever
    while(true) {

      // wait till we get the next message
      Message m = receiveMessage();

      // dispatch message to correct handler
      switch(m.sender()) {
        case 1: h1.handle(m); break;
        case 2: h2.handle(m); break;
        case 3: h3.handle(m); break;
      }
    }
  }
}
```

```
/* ADDITIONAL DEFINITIONS USED IN QUESTION 5 */

class Dispatcher {
  Map<Integer, List<Handler>> registry;

  Dispatcher() {
    registry = new HashMap<Integer, List<Handler>>();
  }

  void register(int sender, Handler h) {
    List<Handler> hs = registry.get(sender);
    if(hs == null)
        hs = new ArrayList<Handler>();
    hs.add(h);
    registry.put(sender, hs);
  }

  /* remove h from the handlers listening for messages from sender */
  void unregister(int sender, Handler h) {
    registry.remove(sender);
  }

  void dispatch(Message m) {
    List<Handler> hs = registry.get(m.sender());
    for(Handler h : hs)
      h.handle(m);
  }
}

class Listener {
  Dispatcher d;

  Listener(Dispatcher d) {
    this.d = d;
  }

  void listen() {
    while(true)
      d.dispatch(receiveMessage());
  }
}
```