CSE 331 Mock Midterm Exam, Autumn 2026

Tuesday, November 10, 2026

Name	
UW email	

This exam contains 16 pages (including this cover page) and 7 questions, each worth 24 points. Let the staff know if any pages are missing or out of order.

Write clearly. Write each answer inside its box. To change an answer, cross it out completely and write the new one clearly beside it, inside the box. An answer that could be read more than one way, or that cannot be read, earns 0 for its whole question, at the staff's discretion.

- You have 50 minutes.
- Have your UW ID ready to be checked.
- You may use one letter-size sheet of handwritten notes, both sides. Otherwise the exam is closed notes.
- Place all electronics out of sight and out of reach.
- Work outside the answer boxes. The work pages and scratch paper are not graded. Raise your hand if you need more scratch paper.

Question 1: Specification Strength

Consider four possible specifications for a method `static void storeLargest(List<Integer> list)`. Every call passes a non-null `list` with at least one element. In the specifications, `list0` is `list` as it was when the method was called, and a list is in *non-decreasing order* when each of its elements is at most the next.

```
Spec A
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list;
            the other elements of list are unspecified

Spec B
  @requires list is in non-decreasing order
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list;
            the other elements of list are unspecified

Spec C
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list,
            leaving the other elements unchanged

Spec D
  @requires no two elements of list are equal
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list,
            leaving the other elements unchanged
```

For each pair, write in its box:

- **S** if the first specification is stronger than the second
- **W** if the first is weaker than the second
- **I** if the two are incomparable

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** **(S / W / I)**
a. A and B	
b. A and C	
c. B and C	

Question 1: Specification Strength, continued

Repeated for your convenience:

> For each pair, write in its box:
>
> - **S** if the first specification is stronger than the second
> - **W** if the first is weaker than the second
> - **I** if the two are incomparable

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** **(S / W / I)**
d. A and D	
e. C and D	
f. B and D	

Question 2: Test Coverage: 3n + 1

Here is a method, with its specification in Javadoc comments. The code correctly implements the specification. You may assume no arithmetic here ever overflows.

```
/**
 * Applies exactly steps updates to n, even if n reaches 1.
 * Each update divides an even n by 2, or replaces an odd n
 * with 3 * n + 1.
 *
 * @requires 0 < n and 0 <= steps
 * @return the value after exactly steps updates
 */
static long collatz(long n, int steps) {
    for (int i = 0; i < steps; i++) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    return n;
}
```

Each row lists a set of test calls. For each row, answer each coverage question about its tests:

- **Y** for yes
- **N** for no

Write only your answer in each box, clearly; work anywhere outside them.	**Statement coverage?** (Y / N)	**Branch coverage?** (Y / N)	**Loop coverage?** (Y / N)

a. `collatz(1, 0)`
`collatz(2, 1)`
`collatz(4, 2)`			

b. `collatz(1, 0)`
`collatz(2, 1)`
`collatz(1, 5)`			

c. `collatz(1, 5)`			

d. `collatz(3, 2)`
`collatz(3, 0)`
`collatz(3, 1)`			

Question 3: Hoare Triples

Each row is a Hoare triple. All variables have type `int`. You may assume no arithmetic here ever overflows.

For each triple, write in its box:

- **T** if the triple is valid
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Valid?** **(T / F)**

a. `{{ x + y = 10 }}`
`x = 0;`
`{{ y = 10 }}`	

b. `{{ 0 <= x }}`
`if (x < 0) {`
`  y = -1;`
`} else {`
`  y = x + 1;`
`}`
`{{ 0 < y }}`	

c. `{{ x + y = n }}`
`x = x - y;`
`y = 2 * y;`
`{{ x + y = n }}`	

d. `{{ true }}`
`if (x == 0) {`
`  y = 0;`
`} else {`
`  y = 1;`
`}`
`x = 0;`
`{{ y = 0 }}`	

Question 3: Hoare Triples, continued

Repeated for your convenience:

> Each row is a Hoare triple. All variables have type `int`. You may assume no arithmetic here ever overflows.
>
> For each triple, write in its box:
>
> - **T** if the triple is valid
> - **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Valid?** **(T / F)**

e. `{{ 0 <= x }}`
`if (0 < x) {`
`  y = x;`
`}`
`{{ 0 <= y }}`	

f. `{{ x = 0 or y = 0 }}`
`if (x != 0) {`
`  if (y != 0) {`
`  x = 1;`
`  y = 1;`
`  }`
`}`
`{{ x = 0 or y = 0 }}`	

g. `{{ 0 <= x }}`
`while (0 < x) {`
`  x = x - 2;`
`}`
`{{ x = 0 }}`	

h. `{{ x != y }}`
`while (x != y) {`
`  int t = x;`
`  x = y;`
`  y = t;`
`}`
`{{ false }}`	

Question 4: Loop Checks: Count Positive Elements

Arrays are non-null. `n` is the length of `a` and does not change. `a[l..r]` is the slice starting at `l` and ending just before `r`. Every numeric variable and array element is an `int`. You may assume no arithmetic here ever overflows. The number of positive values in an empty slice is 0.

```
{{ i = 0 and count = 0 }}
{{ Inv: I }}
while (i < n) {
    if (0 < a[i]) {
        count = count + 1;
    }
    i = i + 1;
}
{{ count = number of positive values in a }}
```

Each row is a candidate loop invariant `I`. With `P` the precondition on the line above `Inv`, `c` the loop's condition, and `Q` the postcondition just after the loop, the rule for a while loop asks three checks of `I`:

- **Check (1):** `P` implies `I`.
- **Check (2):** `{{ I and c }}` body `{{ I }}` is valid.
- **Check (3):** `I and not c` implies `Q`.

Every check may use the stated array shapes and unmodified inputs; only check (1) may use `P`. Checks (2) and (3) start from any state where their claims are true, even one the loop never reaches.

For each candidate `I`, write in each check's box:

- **Y** if the check passes
- **N** if it fails

Write only your answer in each box, clearly; work anywhere outside them.	**Check (1)** **(Y / N)**	**Check (2)** **(Y / N)**	**Check (3)** **(Y / N)**
a. `{{ 0 <= i <= n and 0 <= count <= i }}`			
b. `{{ 0 <= i <= n and count = number of positive values in a[0..i] }}`			
c. `{{ 0 <= i <= n and count = 1 + number of positive values in a[0..i] }}`			
d. `{{ i = 0 and count = 0 }}`			

Question 5: Representation Invariants and Abstraction Functions

Here is a class `Line`, with its specification in Javadoc comments. You may assume no arithmetic here ever overflows.

```
/**
 * An ordered line of numbered tickets at a deli counter, as well as
 * peak, which tracks the most tickets ever waiting at once.
 *
 * Example sequence, showing abstract state (tickets front first):
 *   new Line()                 -> tickets [],     peak 0
 *   take() -> 1; take() -> 2   -> tickets [1, 2], peak 2
 *   serve()                    -> tickets [2],    peak 2
 *   serve(); serve()           -> tickets [],     peak 2
 *   take() -> 3                -> tickets [3],    peak 2
 */
class Line {
    private int issued;   // RI: ...
    private int served;   // AF(this) = ...
    private int peak;

    /** @effects makes a line with no tickets waiting and peak 0 */
    Line() { issued = 0; served = 0; peak = 0; }

    /**
     * @modifies this
     * @effects adds the next ticket at the back:
     *     ticket 1 first, then ticket 2, and so on
     * @return the number of the ticket added
     */
    int take() {
        issued = issued + 1;
        peak = Math.max(peak, issued - served);
        return issued;
    }

    /**
     * @modifies this
     * @effects removes the ticket at the front,
     *     or does nothing if no ticket is waiting
     */
    void serve() { if (served < issued) { served = served + 1; } }

    /** @return the number of tickets waiting */
    int waiting() { return issued - served; }

    /** @return the peak */
    int peak() { return peak; }
}
```

Part 1

For each claim, write in its box:

- **T** if it is true of every `Line` whenever no method of the class is running
- **F** if not

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** **(T / F)**
a. `served <= issued`	
b. `peak = issued - served`	
c. `issued - served <= peak`	
d. `peak = issued`	
e. `peak <= issued`	

Part 2

Which is the abstraction function? Write its letter in the box. In each option the peak is `peak`, and the tickets waiting, front first, are those numbered:

- **A.** `1` to `issued - served`
- **B.** `served` to `issued - 1`
- **C.** `served + 1` to `issued`
- **D.** `issued` down to `served + 1`

When `served = issued`, each option lists no tickets.

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** **(A / B / C / D)**
f. `AF(this)`	

Question 6: Subtypes: Polygons

For each class below, decide whether it is a **behavioral subtype** of the interface it implements: every object must satisfy the interface's specification, starting when its constructor returns. All six classes are Java subtypes of their declared interfaces.

Assume all lengths are positive, finite numbers. Clients call methods and do not access fields directly. Treat `double` calculations as exact real arithmetic. You may assume no arithmetic here ever overflows.

Part 1

Consider this interface:

```
/** A convex polygon with equal sides and equal interior angles. */
interface RegularPolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `RegularPolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype? (T / F)**

a.
`class Square implements RegularPolygon {`
`  private double s;`
`  Square(double s) { this.s = s; }`
`  public double getArea() { return s * s; }`
`  public double getPerimeter() { return s * 4; }`
`}`	

b.
`class RightTriangle implements RegularPolygon {`
`  private double a, b; // the two legs meeting at the right angle`
`  RightTriangle(double a, double b) {`
`  this.a = a; this.b = b;`
`  }`
`  public double getArea() { return a * b / 2; }`
`  public double getPerimeter() {`
`  return a + b + Math.sqrt(a * a + b * b);`
`  }`
`}`	

Question 6: Subtypes: Polygons, continued

Part 2

Now consider this interface.

```
/** A convex polygon whose side lengths can be changed. */
interface ResizablePolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
    /**
     * @requires sides is non-null and lists, in order, the positive
     *     side lengths of some convex polygon
     * @modifies this
     * @effects changes this to a convex polygon with those sides,
     *     even if the number of sides differs from before
     */
    void setSides(double[] sides);
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `ResizablePolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype?** (T / F)

c. `class ResizableSquare implements ResizablePolygon {`
`  private double s;`
`  ResizableSquare(double s) { this.s = s; }`
`  public double getArea() { return s * s; }`
`  public double getPerimeter() { return s * 4; }`
`  public void setSides(double[] sides) { s = sides[0]; }`
`}`	

Question 6: Subtypes: Polygons, continued

Repeated for your convenience:

> **Part 2**
>
> Now consider this interface.
>
> ```
> /** A convex polygon whose side lengths can be changed. */
> interface ResizablePolygon {
>     /** @return this polygon's area */
>     double getArea();
>     /** @return this polygon's perimeter */
>     double getPerimeter();
>     /**
>      * @requires sides is non-null and lists, in order, the positive
>      *     side lengths of some convex polygon
>      * @modifies this
>      * @effects changes this to a convex polygon with those sides,
>      *     even if the number of sides differs from before
>      */
>     void setSides(double[] sides);
> }
> ```
>
> For each class, write in its box:
>
> - **T** if it is a behavioral subtype of `ResizablePolygon`
> - **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype? (T / F)**

d. `class ResizableRightTriangle implements ResizablePolygon {`
`  private double a, b; // the two legs meeting at the right angle`
`  ResizableRightTriangle(double a, double b) {`
`  this.a = a; this.b = b;`
`  }`
`  public double getArea() { return a * b / 2; }`
`  public double getPerimeter() {`
`  return a + b + Math.sqrt(a * a + b * b);`
`  }`
`  public void setSides(double[] sides) {`
`  if (sides.length != 3 ||`
`  sides[0] * sides[0] + sides[1] * sides[1]`
`  != sides[2] * sides[2]) {`
`  throw new IllegalArgumentException("invalid");`
`  }`
`  a = sides[0]; b = sides[1];`
`  }`
`}`	

Question 6: Subtypes: Polygons, continued

Part 3

Finally, consider this interface:

```
/** A convex polygon with equal sides and equal interior angles. */
interface ResizableRegularPolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
    /**
     * @requires 0 < len
     * @modifies this
     * @effects sets every side's length to len, keeping the number
     *     of sides and keeping the polygon regular
     */
    void setSideLength(double len);
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `ResizableRegularPolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype? (T / F)**
e. `class ResizableSquare2 implements ResizableRegularPolygon { private double s; ResizableSquare2(double s) { this.s = s; } public double getArea() { return s * s; } public double getPerimeter() { return s * 4; } public void setSideLength(double sideLength) { s = sideLength; } }`	
f. `class ResizableRightTriangle2 implements ResizableRegularPolygon { private double a, b; // the two legs meeting at the right angle ResizableRightTriangle2(double a, double b) { this.a = a; this.b = b; } public double getArea() { return a * b / 2; } public double getPerimeter() { return a + b + Math.sqrt(a * a + b * b); } public void setSideLength(double len) { a = len; b = len; } }`	

Question 7: Implementing Equality for a Polyline

A final class `Polyline` represents an immutable, ordered sequence of points. Two polylines are equal exactly when their point sequences have the same size and equal points at every index. Adding a repeated point changes the abstract value, even if the line traces the same path.

The class has two fields:

- `List<Point> points` : at least two non-null, immutable points. Repeated points are allowed. Each point has integer coordinates; `Point.equals` compares both coordinates.
- `double total_length` : the sum of the Euclidean distances between consecutive points. The constructor always computes this finite value from the list, starting at 0.0 and adding distances from left to right. It is never stale.

The constructor copies the list. No method exposes or changes it. You may assume no arithmetic here ever overflows.

`List.equals` compares sizes and then corresponding elements with `equals` . `get(i)` throws `IndexOutOfBoundsException` unless `0 <= i` and `i < size()` . The class uses this hash function:

```
    @Override
    public int hashCode() {
        return points.size();
    }
```

Each row replaces the comment in this method:

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Polyline)) return false;
    Polyline other = (Polyline) o;
    // Insert the proposed body here.
}
```

Consider proposals independently. A good method must terminate normally, return true exactly for a `Polyline` with the same abstract value, satisfy Object's equals contract, and leave both objects unchanged.

For each proposed implementation, write in its box:

- **T** if it is a good equals method for this Polyline specification
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Good equals? (T / F)**
a. `return points.equals(other.points);`	

b.
`int n = Math.min(points.size(), other.points.size());`
`for (int i = 0; i < n; i++) {`
`  if (!points.get(i).equals(other.points.get(i))) return false;`
`}`
`return true;`	

c. `return hashCode() == other.hashCode();`	

d.
`return points.equals(other.points)`
`  && total_length == other.total_length;`	

This page is room to work. It is not graded. Keep it attached to your exam.