CIS 1200 Final Exam   May 7, 2025

Name: ______________________________

PennKey (penn login, e.g., `sweirich`): ______________________________

PennID (the “numbers”, e.g., `12001200`): ______________________________

I certify that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this examination.

Signature: ______________________ Date: ______________

- Please wait to begin the exam until you are told it is time for everyone to start.
- When you begin, start by writing your PennKey at the bottom of all the odd-numbered pages in the rest of the exam.
- There are 120 total points. The time for the exam is 120 minutes.
- You may use one letter-sized, two-sided, handwritten sheet of notes during the exam.
- For coding problems, aim for accurate syntax, but we will not grade your code for indentation, spacing, etc.
- There are 19 pages in the exam and an appendix for your reference. Do not write any answers in the appendix as they will not be graded.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- If you need extra space for an answer, you may use the scratch page at the end of the exam; make sure to clearly indicate that you have done this in the normal answer space for the problem.
- Good luck!

,

1. **OCaml List Recursion and Tests** (16 points total)

For each of the following OCaml "mystery" functions below, provide a short text that describes its result. Then, complete the provided test with the result of the function on a sample input, define two additional test cases, and answer whether the function is tail recursive. You will be graded on the naming, correctness, and differentness of your test cases.

1.1 (8 points)

```
let rec strange (lst : int list) : bool =
  begin match lst with
  | [] -> true
  | [h] -> true
  | h1 :: h2 :: t -> (h1 <= h2) && strange (h2 :: t)
  end
```

(a) Short description of the function's result:

(b) Provided test case:

```
let test () =

   strange [1;2;3] =
;; run_test "provided test" test
```

(c) Two additional tests, with informative names:

```
let test () =

;; run_test                                        test
let test () =

;; run_test                                        test
```

(d) Is `strange` tail recursive?

□ Yes □ No

1.2 (8 points)

```
let rec cryptic (lst : ’a list) : (’a list * ’a list) =
  begin match lst with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | h1 :: h2 :: t ->
    let (evens, odds) = cryptic t in
    (h1 :: evens, h2 :: odds)
  end
```

(a) Short description of the function’s result:

(b) Provided test case:

```
let test () =
   cryptic [1;2;9] =
;; run_test "provided test" test
```

(c) Two additional tests, with informative names:

```
let test () =

;; run_test                                         test

let test () =

;; run_test                                         test
```

(d) Is `cryptic` tail recursive?

□ Yes □ No

2. **Binary Search Tree Invariant** (10 points total)

The Java `TreeMap` class is implemented using a Binary Search Tree. This class maintains the Binary Search Tree invariant, storing the entries in the tree in order sorted by the keys.

Based on your understanding of BSTs, check "Yes" if the following methods of this class should make use of the BST invariant for efficient implementation and "No" otherwise.

2.1 `boolean containsKey(Object key)`
Returns true if this map contains a mapping for the specified key.

□ Yes □ No

2.2 `boolean containsValue(Object value)`
Returns true if this map maps one or more keys to the specified value.

□ Yes □ No

2.3 `K firstKey()`
Returns the first (smallest) key currently in this map.

□ Yes □ No

2.4 `V get(Object key)`
Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

□ Yes □ No

2.5 `int size()`
Returns the number of key-value mappings in this map.

□ Yes □ No

2.6 `V put(K key, V value)`
Associates the specified value with the specified key in this map.

□ Yes □ No

2.7 `Collection<V> values()`
Returns a Collection view of the values contained in this map.

□ Yes □ No

2.8 `V remove(Object key)`
Removes the mapping for this key from this TreeMap if present.

□ Yes □ No

2.9 `Map.Entry<K,V> higherEntry(K key)`
Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.

□ Yes □ No

2.10 `void clear()`
Removes all of the mappings from this map.

□ Yes □ No

3. **Java Concepts** (10 points total)

Indicate whether the following statements are True or False. (See Appendix A for documentation of the `Iterator` interface.)

3.1 True □ False □
An object of a subclass type can be assigned to a variable of its superclass type without explicit casting.

3.2 True □ False □
You can instantiate a class or interface directly using the **`new`** keyword.

3.3 True □ False □
Checked exceptions must be either declared in a method signature or caught.

3.4 True □ False □
A `HashMap` can have multiple keys that are null.

3.5 True □ False □
If you try to access an element at an index that is out of bounds of the array, a `NullPointerException` will be thrown.

3.6 True □ False □
Once an array in Java is created, its size cannot be changed.

3.7 True □ False □
You can compare whether two arrays are references to the same location in the heap using the == operator.

3.8 True □ False □
If `arr` is a 2D array, then `arr.length` returns the number of rows and columns.

3.9 True □ False □
You can assign an `Iterator<Object>` to an `Iterator<String>` variable.

3.10 True □ False □
The type parameter for an `Iterator<String>` ensures that the `next()` method returns a `String`.

4. **Java Subtyping** (16 points total)

The next questions refer to the definitions in Appendix B.

4.1 (3 points)

```
_______ cherryBlossom = new JapaneseCherryBlossom();
cherryBlossom.scientificName();
```

Which type(s) could we use for the blank above? (Check all that apply.)

□ `Object`

□ `Plant`

□ `FloweringPlant`

□ `Sunflower`

□ `JapaneseCherryBlossom`

□ `CherryBlossom`

□ *None of the above*

4.2 (3 points)

Suppose that we implement a method:

```
public static void water(CherryBlossom c) {
  // implementation
}
```

Which identifiers (`o`, `p`,`j`, `f`, `c` or `n`) as defined below could be used as arguments to the `water` method? (Check all that apply.)

□ `Object o = new JapaneseCherryBlossom();`

□ `Plant p = new CherryBlossom();`

□ `JapaneseCherryBlossom j = new JapaneseCherryBlossom();`

□ `FloweringPlant f = new Sunflower();`

□ `CherryBlossom c = new JapaneseCherryBlossom();`

□ `CherryBlossom n = null;`

□ *None of the above*

4.3 (2 points) What happens when this code snippet is run? Select one answer.

```
JapaneseCherryBlossom f = new JapaneseCherryBlossom();
System.out.println(f.scientificName());
```

□ `"Prunus"` is printed to the console.

□ `"Prunus serrulata"` is printed to the console.

□ The code is ill typed and doesn't compile.

4.4 (2 points) What happens when this code snippet is run? Select one answer.

```
CherryBlossom f = new JapaneseCherryBlossom();
System.out.println(f.scientificName());
```

□ `"Prunus"` is printed to the console.

□ `"Prunus serrulata"` is printed to the console.

□ The code is ill typed and doesn’t compile.

The next two questions refer to this code snippet.

```
____(a)____ p  = new __(b)____();
System.out.print(p.growingSeason());
```

4.5 (3 points)

What types could appear in blank `(a)`?

□ `Object`

□ `Plant`

□ `FloweringPlant`

□ `Sunflower`

□ `JapaneseCherryBlossom`

□ `CherryBlossom`

□ *None of the above*

4.6 (3 points) What class(es), used in blank `(b)`, would cause this program to output `"Spring"`?

□ `Object`

□ `Plant`

□ `FloweringPlant`

□ `Sunflower`

□ `JapaneseCherryBlossom`

□ `CherryBlossom`

□ *None of the above*

5. **Inheritance, Overriding, and Exceptions** (15 points total)

Consider the Java classes shown in Appendix C. Note that potential `@Override` keywords and **`throws`** declarations have been intentionally omitted from this code. Some method definitions are not shown.

5.1 (3 points) Which of the following are an example of *overriding*? (Select all that apply.)

- □ `CamelGamesMember(String name, int id)` of `CamelGamesMember`
- □ `equals(Object o)` of `CamelGamesMember`
- □ `speak(boolean lie)` of `FrontCamel`
- □ `playGame()` of `FrontCamel`
- □ `ally(CamelGamesMember other)` of `Player`

5.2 (3 points) Which of the following methods are an example of method *overloading*? (Select all that apply.)

- □ `playGame()` of `CamelGamesMember`
- □ `equals(Object o)` of `CamelGamesMember`
- □ `speak(boolean lie)` of `FrontCamel`
- □ `playGame()` of `FrontCamel`
- □ `ally(CamelGamesMember other)` of `Player`

5.3 (3 points) Consider the following snippet of code. For reference, documentation for `contains` appears in Appendix A.

```
FrontCamel f = new FrontCamel("Wei Rich");
Player p = new Player("Oh Caml", 1);
p.ally(f);
p.ally(p);
```

What is the value of each expression after this code executes? (Check one per line.)

`p.allies.contains(f)`

□ **true** □ **false**

`p.allies.contains(p)`

□ **true** □ **false**

`p.allies.contains(new FrontCamel("Wei Rich"))`

□ **true** □ **false**

`p.allies.contains(new Player("Wei Rich", 1))`

□ **true** □ **false**

5.4 (3 points) Suppose we add the following method to the `FrontCamel` class. Note that `mystery()` is a function that might throw an `IllegalArgumentException` and `BufferedReader` and `FileReader` might throw an `IOException`. `IllegalArgumentException` is a subclass of `RuntimeException` and `IOException` is a subclass of `Exception`.

```
public void lookupPreviousGames()  {
  BufferedReader br = new BufferedReader(new FileReader("previous.txt"));
  String game = br.readLine();
  System.out.println("That was my favorite game!");
  br.close();
  mystery(game);
}
```

What exception(s) should `lookupPreviousGames()` indicate that it throws in its method header, if any? (Select all that apply.)

□ `IOException`

□ `IllegalArgumentException`

□ `NullPointerException`

□ None

5.5 (3 points) Now suppose we add the following method to class `Player`. What exception(s) should `panic()` indicate that it throws in its method header, if any? (Select all that apply.)

```
public boolean panic(int i) {
       panic(i + 1);
       return true;
}
```

□ `StackOverflowError`

□ `NullPointerException`

□ `IOException`

□ None

6. **Collections and Iterators** (40 points total)

Recall the type definitions for linked deques in OCaml from HW 4, shown in Appendix D. We can implement a similar data structure in Java using the following classes:

```
class DQNode<E> {
    public final E v;
    public DQNode<E> next = null;
    public DQNode<E> prev = null;
    public DQNode(E value) {  this.v = value; }
}

class LinkedDeque<E> {
    private DQNode<E> head = null;
    private DQNode<E> tail = null;

    // ... see Appendix D for more
}
```

Step 1: Define the deque invariant

6.1 (4 points) Complete the following definition of the deque invariant by filling in each box with a short **Java expression**. The first has been done for you as an example. For reference, the OCaml deque invariant appears in Appendix D.

A `LinkedDeque<E>` is valid when:

- The deque is empty and `head == null && tail == null`, or
- the deque is non-empty and,
  (a) `tail` is reachable from `head` by following `next` pointers
  (b) (there is no element after the tail)
  (c) `head` is reachable from `tail` by following `prev` pointers
  (d) (there is no element before the head)
- For every node `n` in the deque, if `n.next` is not `null`, then
  (e)
- For every node `n` in the deque, if `n.prev` is not `null`, then
  (f)

Step 2: Preserve the deque invariant A partial implementation of the `LinkedDeque` class appears in Appendix D.

6.2 (3 points) Does the following implementation of the `addFirst` method preserve the deque invariant? Check "Yes" or "No".

```
/**
 * insert an element at the beginning of the sequence
 */
public void addFirst(E e) {
    DQNode<E> newNode = new DQNode<E>(e);
    if (head != null) {
        head.prev = newNode;
        newNode.next = head;
    }
    head = newNode;
}
```

□ Yes □ No

If "No", provide a short explanation.

For each of the methods below, decide whether it can be safely added to this class, or if it could be used to violate the deque invariant. If the method is *unsafe*, write a short snippet of code that could be added at the location marked `// HERE` below to violate the invariant.

```
// in a method not part of the LinkedDeque class
LinkedDeque<String> l = new LinkedDeque();
l.addLast("a");
l.addLast("b");
// HERE
```

6.3 (3 points)

```
    public DQNode<E> links() {
        return this.head;
    }
```

□ Safe □ Unsafe

If Unsafe, code that violates invariant

6.4 (3 points)

```
    public LinkedDeque<E> export() {
        return this;
    }
```

□ Safe □ Unsafe

If Unsafe, code that violates invariant

Step 3: Write tests Complete the following JUnit tests based on your understanding of what an iterator for the `LinkedDeque` class should do.

6.5 (5 points)

```
@Test
void testIterator() {
   LinkedDeque<String> deque = new LinkedDeque<>();
   deque.addLast("a");
   deque.addLast("b");
   Iterator<String> it = deque.iterator();

   assertEquals(                    , it.hasNext());

   assertEquals(                    , it.next());

   assertEquals(                    , it.next());

   assertEquals(                    , it.hasNext());

   assertEquals(                    , deque.isEmpty());
}
```

6.6 (2 points)

```
@Test
void testEmptyIterator() {
   LinkedDeque<String> deque = new LinkedDeque<>();
   Iterator<String> it = deque.iterator();

   assertEquals(                    , it.hasNext());
   assertThrows(NoSuchElementException.class,

     () ->                                       );
}
```

6.7 (12 points) Complete the implemention of the `iterator()` method required by the `Iterable` interface. You must finish the definition of the *nested inner class* `DequeIterator`, which is defined inside the `LinkedDeque` class. For reference, `Iterator` appears in Appendix A.

```
@Override
public Iterator<E> iterator() { return new DequeIterator<E>(); }

// nested inner class
private class DequeIterator implements Iterator<E> {
  // instance variables



  // constructor
  public DequeIterator() {



  }
  // methods of Iterator interface



}
```

Step 5: Implement contains Suppose you would like to add the following method to the `LinkedDeque` class, similar to the `contains` method from Java's `Collection` interface (Appendix A).

```
public boolean contains(Object o)
```

Returns true if this deque contains the specified element. More formally, returns true if and only if this deque contains at least one element `e` such that if `o` is `null` then `e` is `null`, otherwise `o.equals(e)`.

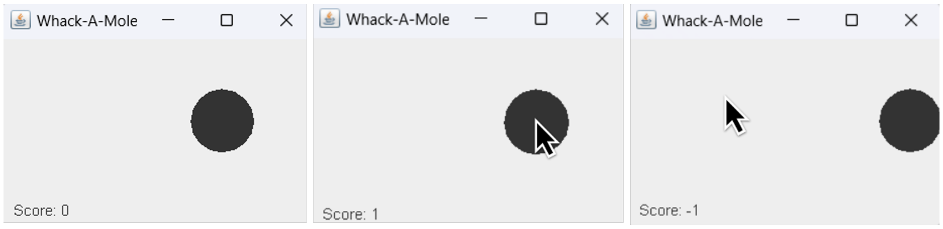
6.8 (8 points) Implement the `contains` method as part of the `LinkedDeque` class. You may assume that the methods shown in Appendix D, including `iterator`, have already been correctly implemented.

```
public boolean contains(Object o) {

}
```

7. **Java Swing Programming** (13 points total)

Appendix F implements a simple Whack-A-Mole game in which circle randomly generates at various positions on the screen. Clicking on the circle increments the score by 1 point, while clicking outside the circle decreases the score by 1 point. The images below show the GUI at the start of a game, after a correct click, and after an incorrect click.



Score: 0

Score: 1

Score: -1

7.1 (2 points) If we removed the call to `repaint` on line 16 of `Whackamole`, what would happen? (Select all that apply.)

- □ The program would not visually update.
- □ `pos` would not update with each interval of the timer.
- □ `score` would not update with mouse clicks.
- □ No change

7.2 (2 points) On which line(s) do we create an instance of an anonymous inner class, including lambda expressions? (Select all that apply.)

- □ Line 3 of `GameRunner`
- □ Line 14 of `Whackamole`
- □ Line 19 of `Whackamole`
- □ Line 49 of `Whackamole`

7.3 (2 points) Which of the following are static method calls? (Select all that apply.)

- □ `SwingUtilities.invokeLater` on line 3 of `GameRunner`
- □ `super.paintComponent` on line 49 of Whackamole
- □ `g.fillOval` on line 51 of Whackamole
- □ `Math.random` on line 30 of Whackamole

7.4 (2 points) What would change if we removed `updatePos()` on line 13 of `Whackamole`? (Select all that apply.)

- □ The circle would not move during game play.
- □ The circle would always start on the far left of the screen.
- □ The circle would not be visible on the screen at any point during game play.
- □ No change.

7.5 (5 points) Now let's add a button to the game that allows us to "level up." Here, leveling up means that each click should be worth an extra point. In other words, after the first time the button is pressed, clicking the mole correctly earns 2 points and a misclick loses 2 points. Pressing the button again increments this to 3 points, and so on. The documentation for relevant parts of the Swing library appears in Appendix E.

Complete the following code, which should be inserted at line 25 in the `Whackamole` class.

```
// create a new button that when pressed, increases the amount
// that score will increment/decrement by 1 point
//
                          levelUpButton =
add(levelUpButton);
```

Scratch Space

Use this page for work that you do not want us to grade. If you run out of space elsewhere in the exam and you ***do*** *want to put something here that we should grade, make sure to put a clear note in the normal answer space for the problem in question.*

A Java Collections Framework interfaces

```
interface Iterator<E> {

    /** Returns true if the iteration has more elements. */
    boolean hasNext()

    /** Returns the next element in the iteration.
        Throws NoSuchElementException if the iteration has no more elements. */
    E next()

    ...
}

interface Iterable<E> {
    /** Returns an iterator over elements of type E. */
    Iterator<E> iterator();


    ...
}

interface Collection<E> extends Iterable<E> {

  /** Returns true if this collection contains the specified element. More
  formally, returns true if and only if this collection contains at
  least one element e such that if o is null then e is null,
  otherwise o.equals(e). */

  boolean contains(Object o)

  // ... other operations
}
```

B Java Code for Subtyping

```java
interface Plant {
    String commonName();
    Boolean hasSeeds();
}

abstract class FloweringPlant implements Plant {
    @Override
    public Boolean hasSeeds() { return true; }

    public abstract String growingSeason();
}

class Sunflower extends FloweringPlant {
    @Override
    public String commonName() { return "Sunflower"; }

    @Override
    public String growingSeason() {  return "Annual"; }
}

class CherryBlossom extends FloweringPlant {
    @Override
    public String commonName() { return "Cherry Blossom";  }

    public String scientificName() { return "Prunus"; }

    @Override
    public String growingSeason() { return "Spring"; }
}

class JapaneseCherryBlossom extends CherryBlossom {
    @Override
    public String commonName() { return "Japanese Cherry Blossom"; }

    @Override
    public String scientificName() { return "Prunus serrulata"; }
}
```

C Java Code for CamelGames (excerpt)

```
class CamelGamesMember {
   private final String name;
   private final int id;

   public CamelGamesMember(String name, int id) {
       this.name = name;
       this.id = id;
   }

   public void speak() { /* ... */ }

   public void playGame() { /* ... */  }

   public boolean equals(Object o) {
       if (o == null || getClass() != o.getClass()) { return false; }
       CamelGamesMember that = (CamelGamesMember) o;
       return id == that.id && ((name == null && that.name == null)
                || name.equals(that.name));
   }
}

class FrontCamel extends CamelGamesMember {
   public FrontCamel(String name) { super(name, 1); }

   public void speak(boolean lie) { /* ... */ }

   public void mystery(String game) { /* ... */ }

   public void playGame() { /* ... */ }
}

class Player extends CamelGamesMember {
   public List<CamelGamesMember> allies = new LinkedList<>();;

   public Player(String name, int id) { super(name, id); }

   public void ally(CamelGamesMember other) {
       this.speak();
       other.speak();
       allies.add(other);
   }

}
```

D OCaml Linked Deque type and invariant

```
type 'a dqnode = {
  v: 'a;
  mutable next: 'a dqnode option;
  mutable prev: 'a dqnode option;
}

type 'a deque = {
  mutable head: 'a dqnode option;
  mutable tail: 'a dqnode option;
}

(* DEQUE invariant: The deque is empty, the head and tail are both None, or
   the deque is non-empty, and
   - head = Some n1 and tail = Some n2, where
      (a) n1 is reachable from n1 by following 'next' pointers
      (b) n2.next = None   (there is no element after the tail)
      (c) n1 is reachable from n2 by following 'prev' pointers
      (d) n1.prev = None   (there is no element before the head)
   - for every node n in the deque:
      (e) if n.next = Some m then
            m.prev = Some n
      (f) if n.prev = Some m then
            m.next = Some n *)
```

E Java Linked Deque

```
class DQNode<E> {
    public final E v;
    public DQNode<E> next = null;
    public DQNode<E> prev = null;
    public DQNode(E value) {  this.v = value; }
}

class LinkedDeque<E> implements Iterable<E> {
    private DQNode<E> head;
    private DQNode<E> tail;

    /** Creates an empty deque */
    public LinkedDeque<E>() { ... }

    /** Returns true if the deque contains no elements. */
    public boolean isEmpty() { ... }

    /** Appends the specified element at the end of this deque. */
    public void addLast(E e) { ... }

    /** Returns an iterator over the elements in this deque, starting with the
    element stored in the head node and continuing to the tail. */
    public Iterator<E> iterator() { ... }

    /** Inner class for iterator */
    private class DequeIterator<E> implements Iterator<E> { ... }

    // other methods not shown
}
```

F Java Swing library (excerpt)

```
interface ActionListener {
  void actionPerformed(ActionEvent e);
}

class JButton {
    public JButton () {
        ...
    }

    public JButton (String text) {
        ...
    }

    public void addActionListener (ActionListener l) {
        ...
    }

    ...
}
```

G Java Code for Whackamole

```java
public class Whackamole extends JPanel {
    private final int BOARD_HEIGHT = 150;
    private final int BOARD_WIDTH = 250;
    private final int NUM_SPOTS = 5;
    private final int BOX_SIZE = BOARD_WIDTH/NUM_SPOTS;
    private int pos = 0;
    private int score = 0;
    private int scoreChange = 1;

    public Whackamole (){
        updatePos();
        Timer timer = new Timer(1000, e -> {
            updatePos();   // code to run every interval
            repaint();
        });
        timer.start();
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                updateScore(e.getX()/BOX_SIZE);
            }
        });

    }

    private void updatePos(){
        this.pos = (int)Math.floor(Math.random() * NUM_SPOTS);
    }

    private void updateScore(int clickPos){
        if (clickPos == this.pos){
            score = score + scoreChange;
        } else {
            score = score - scoreChange;
        }
    }

    @Override
    public Dimension getPreferredSize(){
        return new Dimension (BOARD_WIDTH, BOARD_HEIGHT);
    }

    @Override
    public void paintComponent(Graphics g){
        super.paintComponent(g);
        g.fillOval(pos * BOX_SIZE, 40, BOX_SIZE, BOX_SIZE);
        g.drawString("Score: " + score, 10, BOARD_HEIGHT - 10);
    }
}
```

```
public class GameRunner {
    public static void main (String[] args){
        SwingUtilities.invokeLater(()-> {
                    JFrame f = new JFrame("Whack-A-Mole");
                    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    f.add(new Whackamole());
                    f.pack();
                    f.setVisible(true);
                }
        );
    }
}
```