CIS 1200 Final Exam December 19, 2023

Steve Zdancewic and Swapneel Sheth, instructors

Name: _____

PennKey (penn login id, e.g., `stevez`): _____

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, please 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 exam length is 120 minutes.
- For coding problems: aim for accurate syntax, but we will not grade your code style for indentation, spacing, etc.
- There are 16 pages in the exam and an Appendix for your reference. Please do not submit the Appendix.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- Good luck!

1. **OCaml and Java Concepts** (21 points total)

   (a) (12 points) (1.5 points each) Indicate whether the following statements are true or false.

   (a) True □ False □
   In Java, if `s.equals(t)` returns **`true`**, then `s == t` is guaranteed to return **`true`**.

   (b) True □ False □
   In Java, if `s == t` returns **`true`**, then `s.equals(t)` is guaranteed to return **`true`**.

   (c) True □ False □
   In our GUI library, an `event_listener` is a first-class function stored in the hidden state of a notifier widget. When an event occurs in the widget, the notifier invokes all of the stored `event_listener`s.

   (d) True □ False □
   Variables in OCaml are immutable by default, whereas in Java, variables are mutable by default.

   (e) True □ False □
   In a Java **`try`**-**`catch`**-**`finally`** statement, the **`finally`** clause is executed only if the **`try`** clause does not throw an exception.

   (f) True □ False □
   In Java, a `final` instance variable can only be changed in a constructor.

   (g) True □ False □
   In Java, every method must declare every exception it might throw in its header.

   (h) True □ False □
   In Java, the default implementation of equality in the `Object` class uses reference equality for mutable objects and structural equality for immutable objects.

(b) For the following questions, select the most suitable data structure for each particular use case and justify why.

(Hint: An `ArrayList` in Java is similar to the "Resizable Array" example from class.)

i. (3 points) You are developing a navigation application (similar to Google Maps) where a route consists of a series of way-points. You will need to frequently insert and remove way-points as the route is optimized in real time. Which collection is best suited for this?

☐ `ArrayList` ☐ `LinkedList` ☐ `TreeMap`

Reason:

_____

_____

ii. (3 points) You're designing a system for an online bookstore (similar to Amazon) to store books where they need to be sorted based on ISBN numbers and frequently retrieved (using the ISBN numbers). Which collection would you choose?

☐ `ArrayList` ☐ `LinkedList` ☐ `TreeMap`

Reason:

_____

_____

iii. (3 points) You're designing a system for a music playing app (similar to Spotify) that stores curated playlists and once these playlists are created, they are never updated. Users can choose to play songs in any order they like (e.g., song 1 followed by song 3 followed by song 10). Which collection would you choose?

☐ `ArrayList` ☐ `LinkedList` ☐ `TreeMap`

Reason:

_____

_____

2. **OCaml Higher Order Functions (again)** (16 points total)

Recall the higher-order list processing functions shown in Appendix 1.

For each of the following mystery functions, determine which implementation choice(s) will produce the correct output for the provided `input` list (Choose all that apply).

(a) (4 points)

```
mystery1 [1; 4; 7] = [2; 1; 5; 4; 8; 7]
```

☐
```
fold (fun x acc -> (x+1)::x::acc) [] input
```

☐
```
transform (fun x-> x+1::[x]) input
```

☐
```
fold (fun x acc -> x+1@x@acc) [] input
```

☐
```
transform (fun x -> (x, x+1)) input
```

(b) (4 points)

```
mystery2 [("I", "love"); ("dogs", "and"); ("cats", "!")] =
  ["I love"; "dogs and"; "cats !"]
```

☐
```
fold (fun (a,b) acc -> a^" "^b) [] input
```

☐
```
transform (fun (a,b) -> a^" "^b) input
```

☐
```
fold (fun (a,b) acc -> (a^" "^b)::acc) [] input
```

☐
```
transform (fun (a,b) -> (a^" "^b)::[]) input
```

(c) (4 points)

```
mystery3 [2; 3; 4] = 1
```

☐
```
fold (fun x acc -> acc / x) 24 input
```

☐
```
transform (fun x -> 1) input
```

☐
```
fold (fun x acc ->
  if ((x mod 3 = 0) || (acc = 1)) then 1 else 0) 0 input
```

☐
```
transform (fun x -> if x = 2 then 1 else 0) input
```

(d) (4 points)

```
mystery4 [[1; 2; 3]; [-1; -1; -1]; [2; 2; 2]] = [6; -1; 8])
```

☐
```
transform (fun x -> (fold (fun y acc -> y * acc) 1 x)) input
```

☐
```
transform (fun x -> (transform (fun y -> y * y) x)) input
```

☐
```
fold (fun x1 acc -> (transform (fun x2 -> x1*x2::acc))) [] input
```

☐
```
fold (fun x acc1 ->
  (fold (fun y acc2 -> y * acc2) 1 x)::acc1) [] input
```

3. **Java Subtyping and Dynamic Dispatch** (19 points total)

This problem refers to two interfaces and several classes that might appear in a program about the movie "Barbie". You can find them in Appendix 2.

(a) (2 points) Which of the following classes are an example of simple inheritance in Java (either explicitly or implicitly)? (Mark all that apply.)

☐ `Allan` ☐ `ComputerScienceBarbie` ☐ `JavaBarbie`
☐ `OcamlBarbie`

(b) (2.5 points)

```
__________________ barbie = new ComputerScienceBarbie("CIS", "Hoodie");
```

Which type can be correctly used for the declaration of `barbie` above?
(Mark all that apply.)

☐ `MattelToy`
☐ `Barbie`
☐ `Allan`
☐ `ComputerScienceBarbie`
☐ `JavaBarbie`
☐ `OcamlBarbie`
☐ `Object`

(c) (2.5 points)

```
MattelToy barbie = new _______??_________(...);
OCamlBarbie toy = (OCamlBarbie) barbie;
toy.debugOCamlCode("'a list");
```

What can be used on the first line (instead of the `???`) so that the code successfully compiles *but throws an exception when run*? (Choose all that apply.)

☐ `MattelToy`
☐ `Barbie`
☐ `Allan`
☐ `ComputerScienceBarbie`
☐ `JavaBarbie`
☐ `OcamlBarbie`
☐ `Object`

Which of the following lines is legal Java code that will not cause any compile-time (i.e., type checking) or run-time errors?

If it is legal code, check the "Legal Code" box and answer the questions that follow it. If it is not legal, check one of the "Not Legal" options and explain why.

You can assume each option below is independent.

(d) (3 points)

```
Barbie barbie = new ComputerScienceBarbie("Natalie", "Blazer");
String result = barbie.code();
```

☐ Legal Code

A. The static type of `barbie` is _____.

B. The dynamic class of `barbie` is _____.

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(e) (3 points)

```
MattelToy toy = new OCamlBarbie("OCaml", "Caml");
toy.manufacture();
```

☐ Legal Code

A. The static type of `toy` is _____ .

B. The dynamic class of `toy` is _____ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(f) (3 points)

```
ComputerScienceBarbie barbie = new JavaBarbie("Java", "Coffee cup");
System.out.println(barbie.speak());
```

The code above is Legal and doesn't throw an exception when run. Now, consider changing the second line to the following:

```
System.out.println(((JavaBarbie) barbie).speak());
```

The new version is:

- ☐ Legal Code

  The updated code above will print (Choose all that apply.)

  - ☐ "Hi Computer Science Barbie"
  - ☐ "Hi Java Barbie"
  - ☐ "Hi OCaml Barbie"
- ☐ Not Legal — Will compile, but will throw an `Exception` when run
- ☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

_____

(g) (3 points)

```
List<ComputerScienceBarbie> list = new ArrayList<JavaBarbie>();
```

- ☐ Legal Code

  The code above is legal because: (Choose all that apply.)

  - ☐ `JavaBarbie` is a subtype of `ComputerScienceBarbie`
  - ☐ This example illustrates Parametric Polymorphism.
- ☐ Not Legal — Will compile, but will throw an `Exception` when run
- ☐ Not Legal — Will not compile

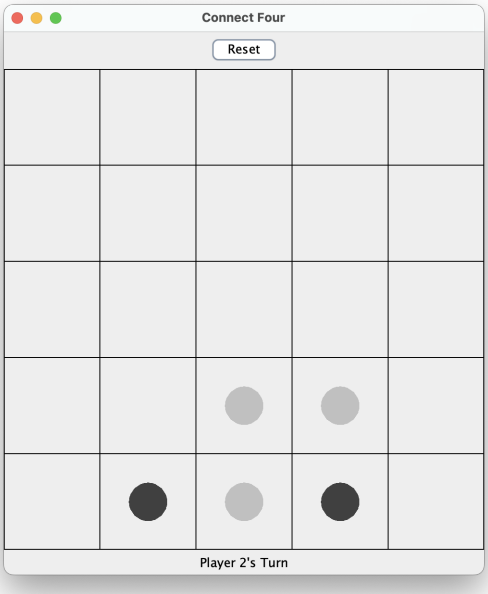
Reason for not legal (in either of the two illegal cases above):

_____

4. **Java Swing Programming** (16 points total)

Appendix 3 shows code for a partial implementation of a "Connect Four" game in Java.

The image below shows the GUI after a few turns. The exact rules of the game are not relevant to the questions below; the questions test your understanding of both Java and Swing programming idioms instead.



(a) (2 points)

True ☐ False ☐

On line 44, calling `super.paintComponent(g)` invokes the `paintComponent` method of the `JPanel` class.

(b) (2 points)

There are several occurrences of the **`new`** keyword in the code provided. List all occurrences (i.e., line numbers) that correspond to anonymous inner classes.

(c) (4 points)

How will the program's behavior change if we delete the call to `repaint()` on line 31? (Select one.)

- ☐ Nothing at all will ever be displayed – just a blank window.
- ☐ The initial GUI will be displayed, but no shapes will ever be drawn.
- ☐ The initial GUI will be displayed and tiles will show up when the grid is selected, but clicking the reset button will have no effect.
- ☐ The initial GUI will be displayed and tiles will show up when the grid spots are selected. Clicking the reset button will not clear the existing tiles and no further GUI updates will happen (regardless of what the user does).
- ☐ The initial GUI will be displayed and tiles will show up when the grid spots are selected. Clicking the reset button will not clear the existing tiles, but clicking on a grid spot will update the UI to the correct state.
- ☐ No change in behavior.

(d) (4 points) For the game overall, which of the following are true? (Mark all that apply.)

- ☐ `GameBoard` is a supertype of `JPanel`.
- ☐ The code would still compile successfully and behave identically if we replace the Lambda function on lines 74–97 with an analogous anonymous inner class.
- ☐ The `@Override` annotations (such as on lines 16, 42, and 65) are optional and the code will successfully compile if we delete them.
- ☐ Java requires the use of the `MouseAdapter` class whenever we want to add a `MouseListener` to an object.

(e) (4 points) Consider the argument to `reset.addActionListener(...)` on line 90. Which of the following are true?

The Javadocs for `addActionListener` are shown below:

```
public void addActionListener(ActionListener l)
Adds an ActionListener to the button.

Parameters:
l - the ActionListener to be added
```

(Select one.)

- ☐ The static type of the argument is **`null`**.
- ☐ The static type of the argument is `Object`.
- ☐ The static type of the argument is `ActionListener`.
- ☐ The argument has a static type, but it's not one of the options listed above.
- ☐ The argument does not have a static type.

(Select one.)

- ☐ The dynamic class of the argument is **`null`**.
- ☐ The dynamic class of the argument is `Object`.
- ☐ The dynamic class of the argument is `ActionListener`.
- ☐ The argument has a dynamic class, but it's not one of the options listed above.
- ☐ The argument does not have a dynamic class.

5. **Java Exceptions** (12 points total)

This problem makes use of the class definitions for a game called "Hot Potato," which is provided for you in Appendix 4.

(a) (5 points)

The `getsIt` method has a **`throws`** declaration in the method signature (line 14). Which of the following are true? (Choose all that apply.)

- ☐ It is possible to omit the **`throws`** declaration (*without* making any other changes) and the code would still compile successfully.
- ☐ It is possible to omit the **`throws`** declaration (*with* making other changes) and the code would still compile successfully.
- ☐ Any code that calls the `getsIt` method must also have a **`throws`** declaration in the method signature.
- ☐ Any code that calls the `getsIt` method doesn't need to have a **`throws`** declaration in the method signature since `HotPotato` is an *unchecked* exception.
- ☐ Any code that calls the `getsIt` method either needs a **`throws`** declaration in the method signature or handles the exception via a **`try`**–**`catch`** block since `HotPotato` is a *checked* exception.

(b) (7 points)

What happens when the following code is run? (Select one.)

```
String n1 = "Player 1";
String n2 = "Player 2";
String n3 = "Player 3";
HotPotato potato = new HotPotato();
Player p1 = new Player(n1, null);

Player p2 = new Player(n2, null);
Player p3 = new Player(n3, p2);
p1.nextInLine = p3;
p3.nextInLine.nextInLine = p1;

Player.thrower = p3;
try {
    p2.getsIt(potato);
} catch (HotPotato p) {
    System.out.println("Dropped it!");
}
```

(Answer choices on the next page.)

□ Nothing is printed to the console and the program immediately terminates.

□ The program throws a `NullPointerException` (possibly after printing some output).

□ The console prints the following output and execution terminates normally.

```
Player 2 gets the potato.
Player 2 throws the potato.
Player 2 catches the potato and throws it.
Dropped it!
```

□ The console prints the following output and execution terminates normally.

```
Player 2 gets the potato.
Player 2 passes the potato.
Player 1 gets the potato.
Player 1 passes the potato.
Player 3 gets the potato.
Player 3 throws the potato.
Player 1 catches the potato and throws it.
Player 2 catches the potato and throws it.
Dropped it!
```

□ The console prints the following output and execution terminates normally.

```
Player 2 gets the potato.
Player 2 passes the potato.
Player 3 gets the potato.
Player 3 passes the potato.
Player 1 gets the potato.
Player 1 throws the potato.
Player 2 catches the potato and throws it.
Dropped it!
```

□ The console prints

```
Player 2 gets the potato.
Player 2 passes the potato.
Player 1 gets the potato.
Player 1 passes the potato.
Player 3 gets the potato.
Player 3 passes the potato.
Player 2 gets the potato.
Player 2 passes the potato.
Player 1 gets the potato.
Player 1 passes the potato.
```

and continues looping until a `StackOverflow` occurs.

6. **Java Iterators and Testing** (36 points total)

In this problem, you will create a class, `NumberGenerator`, which is an `Iterator` that produces a sequence of numbers for which a provided validator returns **`true`**.

The numbers are non-negative and they are produced in ascending order.

For example, to print out the first 10 prime numbers, we could use the following code:

```
Iterator<Integer> primes = new NumberGenerator(new PrimeNumberValidator());
for(int i = 0; i < 10; i++) {
    System.out.print(primes.next() + " ");
}
```

When executed, the output will be:

```
2 3 5 7 11 13 17 19 23 29
```

In the code above, the `PrimeNumberValidator` class (see Appendix 5) is an instance of the `NumberValidator` interface, which is defined as:

```
public interface NumberValidator {
    public boolean isValidNumber(int x);
}
```

In the case of `PrimeNumberValidator`, the `isValidNumber(x)` method returns **`true`** if `x` is prime, and returns **`false`** otherwise.

(a) (3 points) Fill in the blanks below so that correct implementations of `NumberGenerator` and `PrimeNumberValidator` will pass, *or*, if that is not possible, mark the box indicating why.

```
@Test
public void testNumberGeneratorTestA() {

    int[] primes = {2, 3, __________, 7, 11, 13, 17, 19, 23, 29};

    Iterator<Integer> iter = new NumberGenerator(new PrimeNumberValidator());

    for(int i = 0; i < 5; i++ ) {

      assertEquals(primes[i], ____________________);
    }
    assertEquals(13, iter.next());

    assert____________________(iter.hasNext());
}
```

OR (choose one)

- ☐ It is not possible because of syntax errors in the provided code.
- ☐ It is not possible because of type errors in the provided code.

(b) (3 points) Fill in the blanks in the test case below so that it will pass, *or*, if that is not possible, mark the box indicating why.

```
@Test
public void testNumberGeneratorTestB() {
    Iterator<Integer> iter = new NumberGenerator(x -> x==0);

    assertEquals(____________________, iter.next());

    assert____________________(iter.hasNext());

    assertThrows(_____________________________________________, () -> iter.next());
}
```

OR (choose one)

- ☐ It is not possible because of syntax errors in the provided code.
- ☐ It is not possible because of type errors in the provided code.

Constructors The `NumberGenerator` will support two constructors. The first one accepts an **`int`** *upper bound* with the meaning that all yielded results be *strictly less than* that bound, along with a `NumberValidator`. If the bound provided is negative or the `NumberValidator` is **`null`**, the `NumberGenerator` constructor should throw an `IllegalArgumentException`.

The second constructor (that we provide) accepts just a `NumberValidator` and invokes the first constructor with a bound of `Integer.MAX_VALUE`, which is the largest **`int`** value representable in Java.

(c) (2 points)

True ☐ False ☐

The presence of two constructors for `NumberGenerator` that have the same name but differ in their types is an example of *overriding* in Java.

(d) (3 points) Fill in the blanks in the test case below so that it will pass, *or*, if that is not possible, mark the box indicating why.

```
@Test
public void testC() {
    assertThrows(IllegalArgumentException.class,
                 () -> new NumberGenerator(____________________, x -> x==0));

    assertThrows(IllegalArgumentException.class,
                 () -> new NumberGenerator(3, ____________________));

    Iterator<Integer> iter = new NumberGenerator(2, x -> true);
    assertEquals(0, iter.next());
    assertEquals(1, iter.next());

    assert_________________(iter.hasNext());
}
```

OR (choose one)

- ☐ It is not possible because of syntax errors in the provided code.
- ☐ It is not possible because of type errors in the provided code.

Implementation

The following page contains a partial implementation of the `NumberGenerator` class itself. Complete the code so that it correctly implements the `Iterator<Integer>` interface (see the JavaDocs in Section 6).

After a `NumberGenerator` is constructed from a given `NumberValidator` and `bound`, each call to `next()` should yield the smallest non-negative (i.e. `>= 0`) integer that has not already been yielded and such that `v.isValidNumber(n) == true` and the number is `< bound`. If there is no such number, `next()` should throw a `NoSuchElementException`. The yielded values should be generated on demand, that is: your code should generate at most one number at a time.

You may *not* use any other Java libraries but you can assume that `Iterator` and appropriate exception types are imported.

(e) (4 points) You will have to add at least one **`private`** field and may add private helper method(s). In the space below, briefly state the representation *invariant(s)* used by your code for the private field(s). Hint: the representation invariant should be *exploited* by `hasNext()`.

INVARIANT:

(f) (21 points) Complete this implementation:

```
/** Generates an ascending sequence of numbers all of which
 * are >= 0 and < the given bound and that also satisfy the
 * isValidNumber predicate provided by the NumberValidator.
 */
public class NumberGenerator implements Iterator<Integer> {
    private final NumberValidator v;
    private final int bound;    // all results are < bound
    // TODO: add additional field(s) here




    public NumberGenerator(int bound, NumberValidator v) {
        if (bound < 0 || v == null) throw new IllegalArgumentException();
        this.v = v;
        this.bound = bound;
        // TODO: complete this constructor




    }
    // provided constructor with a large default maximum value
    public NumberGenerator(NumberValidator v) {
        this(Integer.MAX_VALUE, v);
    }
    // TODO: Add helper method(s) here:









    public boolean hasNext() {




    }

    public Integer next() {




    }
}
```