NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 2 AY2017/2018

CS2030 Programming Methodology II

May 2018 Time Allowed 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 17 questions and comprises 16 printed pages, including this page.
2. Write all your answers in the answer sheet provided.
3. The total marks for this assessment is 70. Answer **ALL** questions.
4. This is an **OPEN BOOK** assessment.
5. All questions in this assessment paper use Java 9 unless specified otherwise.
6. State any additional assumption that you make.
7. Please write your student number only. Do not write your name.

Part I
Multiple Choice Questions (36 points)

- For each of the questions below, **write your answer in the corresponding answer box on the answer sheet.** Each question is worth 3 points.

Questions 1 and 2 are based on the following scenario. Consider an object-oriented program written for the game of chess. A game consists of a chess board and multiple chess pieces moving on the chess board. There are six types of chess pieces: king, queen, bishop, rook, knight, and pawn. Each type of chess piece is drawn differently on the screen and each moves on the chess board in a different way. The chess board consists of multiple cells and each cell either contains a chess piece or is empty. There are two players for each game, and each chess piece belongs to either one of the players.

You created the following classes: `ChessBoard`, `ChessPiece`, `King`, `Queen`, `Bishop`, `Knight`, `Rook`, `Pawn`, `Cell`, and `Player` to model the corresponding nouns in the above description.

1. (3 points) Which of the following decision to model the chess game is the LEAST appropriate?

   A. Subclasses of `ChessPiece` override the `draw` method in `ChessPiece`.

   B. Subclasses of `ChessPiece` override the `move` method in `ChessPiece`.

   C. `Cell` contains a field of type `Optional<ChessPiece>`.

   D. `ChessPiece` inherits from `Player`.

   E. `ChessPiece` is an abstract class.

   Write `X` in the answer box if all of the choices above are equally appropriate.

   > **Solution:** D. `ChessPiece` should contain a field of type `Player` instead.
   >
   > This is a give-away OO modeling question to check if you know the basic OO concepts of overriding, HAS-A relationship, IS-A relationship, and abstract class.

2. (3 points) We model `King`, `Queen`, `Bishop`, `Knight`, `Rook`, and `Pawn` as subclasses of `ChessPiece`. Let's say that we want to keep the pieces belong to one player in a `List<...>` called `pieces`. We want to be able to write code like the follows:

   ```
   pieces.add(new Queen());
   ChessPiece p = pieces.remove(0);
   ```

   What should the type of the variable `pieces` be?

   A. `List<>`

   B. `List<?>`

   C. `List<ChessPiece>`

   D. `List<? super ChessPiece>`

   E. `List<? extends ChessPiece>`

   F. `List<King, Queen, Bishop, Knight, Rook, Pawn>`

   Write `X` in the answer box if none of the choices above is correct.

Solution: `C` is the intended answer. Note that `pieces` is both a consumer (`add`) and a supplier (`remove`), applying the PECS principle, it has to both `extends` and `super` the `ChessPiece` class.

In the original question, there is a typo: I wrote `pieces.remove()` instead of `pieces.remove(0)`. Because of this, I will accept `X` as a valid answer.

3. (3 points) Consider the code below:

```
class A {
  static int x;

  static int foo() {
    return 0;
  }

  int bar() {
    return 1;
  }

  static class B {
  }

  public static void main(String[] args) {
    :
  }
}
```

Which of the following line of code, when called from `main`, would result in a compilation ERROR?

A. `x = 1;`

B. `A a = new A();`

C. `B b = new B();`

D. `new A().foo();`

E. `new A().bar();`

Write X in the answer box if none of the choices above would lead to a compilation error.

Solution: X.

The purpose of this question is to check if you understand the implication of `static`.

4. (3 points) Which of the following process happens ONLY during compilation time in Java?

(i) type inference – inferring the type of a variable whose type is not specified.

(ii) type erasure – replacing a type parameter of generics with either Object or its bound.

(iii) type checking – checking if the value matches the type of the variable it is assigned to.

A. Only (i)

B. Only (i) and (ii)

C. Only (i) and (iii)

D. Only (ii) and (iii)

E. (i), (ii), and (iii)

Write X in the answer box if none of the combinations is correct.

Solution: B. Type checking happens during runtime since the value is not always available at compile time.

5. (3 points) Which of the following process happens ONLY during runtime in Java?

   (i) late binding – determine which instance method to call depending on the type of a reference object.

   (ii) type casting – converting the type of one variable to another.

   (iii) accessibility checking – checking if a class has an access to a field in another class.

   A. Only (i)
   B. Only (ii)
   C. Only (i) and (iii)
   D. Only (ii) and (iii)
   E. (i), (ii), and (iii)

   Write X in the answer box if none of the combinations is correct.

   Solution: A. Note that type casting happens during compile time but is checked during run-time. Accessibility checks happen both during run time and compile time.

6. (3 points) Suppose we have two classes `Shape` and `Circle`. `Circle` is a subclass of `Shape`. Recall that `Double` is a subclass of `Number`.

   We declare the following variables:

```
Function<Shape, Double> s2d;
Function<Circle, Number> c2n;
Function<Object, Object> o2o;
```

   Which of the following assignment would result in a compilation error?

   (i) `s2d = c2n;`

   (ii) `o2o = c2n;`

   (iii) `c2n = o2o;`

   A. Only (i)
   B. Only (ii)
   C. Only (i) and (iii)
   D. Only (ii) and (iii)
   E. (i), (ii), and (iii)

   Write X in the answer box if none of the combinations is correct.

   Solution: E. Java Generics are invariant.

   Interestingly, there is a Piazza post related to this, on more complex type inferences for `Function`, just before the exam :)

7. (3 points) Suppose we have a class A that implements the following methods:

```
class A {
  int x;
  boolean isPositive;

  static A of(int x) {
    A a = new A();
    a.x = x;
    a.isPositive = (x >= 0);
    return a;
  }

  A foo(Function<Integer, A> map) {
    return map.apply(this.x);
  }

  A bar(Function<Integer, A> map) {
    if (this.isPositive) {
      return map.apply(this.x);
    } else {
      return A.of(this.x);
    }
  }
}
```

Which of the following conditions hold for A for all values of x. f and g are both variables of type Function<Integer,A>; a is an object of type A.

(i) `A.of(x).foo(f)` always returns `f.apply(x)`

(ii) `a.foo(f).bar(g)` equals to `a.foo(x -> f.apply(x).bar(g))`

(iii) `a.bar(f).bar(g)` equals to `a.bar(x -> f.apply(x).bar(g))`

A. Only (i)

B. Only (ii)

C. Only (i) and (iii)

D. Only (ii) and (iii)

E. (i), (ii), and (iii)

Write X in the answer box if none of the combinations is correct.

Solution: E.

First, let's understand what A does. A wraps around an `int` value x, along with the context of whether the value is positive or not.

The method `foo` simply applies the given method f on the internal value x and returns `f.apply(x)` (map is f). So (i) is true.

The method `bar` is a bit controversial – it selectively applies f on the internal value x. It applies f only if the value is positive, and leave the value untouched otherwise. To check if (ii) and (iii) hold, we can systematically analyze the cases.

Let `x` be the value contained in `a`.

Let's check for (ii).

- `a.foo(f)` is just `f.apply(x)`.
- If `f.apply(x)` is not positive, `a.foo(f).bar(g)` is just `a.foo(f)`. `a.foo(x -> f.apply(x).bar(g))` is just `a.foo(x -> f.apply(x))`, which is just `a.foo(f)`.
- What if `f.apply(x)` is positive? Then `a.foo(f).bar(g)` is `g.apply(f.apply(x))` wrapped in `A`.
- `a.foo(x -> f.apply(x).bar(g))` is also `a.foo(x -> g.apply(f.apply(x)))`.
- Since `foo` applies the given lambda unconditionally, we get `g.apply(f.apply(x))` wrapped in `A` as well.

(ii) holds.

Let's check for (iii). Suppose `x` is positive, then `a.bar(f)` is no different from `a.foo(f)`, the same argument above holds. Suppose `x` is non-positive, then `a.bar(f).bar(g)` is just `a`. `a.bar(x -> f.apply(x).bar(g))` is also just `a` (`a.bar(anything)` is `a`).

So (iii) holds.

Question: Does `a.bar(f).foo(g)` == `a.bar(x -> f.apply(x).foo(g))` hold?

8. (3 points) Consider the following two statements:

```
Stream.of(20, 40, 60, 80, 100, 120, 140)
    .reduce(100, (x, y) -> Math.max(x, y)); // Statement X

Stream.of(20, 40, 60, 80, 100, 120, 140)
    .parallel()
    .reduce(100, (x, y) -> Math.max(x, y)); // Statement Y
```

Which of the following about Statement X and Statement Y above is CORRECT:

A. Statement X may produce a different answer than Statement Y.
B. Statement Y always returns the same value every time it is executed.
C. Statement X returns 100.
D. Statement X returns 20.
E. Statement Y always runs faster than Statement X because the `reduce` operation is executed in parallel.

Write `X` in the answer box if none of the choices is correct.

Solution: B. `Math.max` is associative, so both statements always return the same value (140).

9. (3 points) Wei Tsang tried to demonstrate to the class that parallelizing code with side effects will lead to an undeterministic result. He wrote the following method:

```
void foo() {
  int sum = 0;
  Stream.of(1, 2, 3, 4, 5)
      .parallel()
      .forEach(i -> {
        sum = sum + i;     // Line 6
      });
  System.out.println(sum); // Line 8
}
```

This is not a good demonstration, because:

A. `foo` always prints 15 since `forEach` will be executed sequentially.
B. `foo` always prints 15 since the lambda expression passed to `forEach` has no side effect.
C. `foo` will not compile because Java expects `sum` to be either final or effectively final.
D. `foo` always prints 0 since, due to variable capture, there are two copies of `sum` now, and the captured version of `sum` is being incremented in Line 6.
E. `foo` may print different results every time `foo` is invoked, but not due to side effects. The reason is that `System.out.println` on Line 8 is invoked in parallel with the code on Line 6. The code should call `join()` to wait for all the elements in the stream to be added to `sum` before printing.

Write `X` in the answer box if none of the choices is correct.

Solution: C.

This question tests your understanding about parallel streams and variable capture at the same time.

10. (3 points) Consider the following `RecursiveTask` called `BinSearch` for finding an item within a sorted array using binary search.

```
class BinSearch extends RecursiveTask<Boolean> {
  int low;
  int high;
  int toFind;
  int[] array;

  BinSearch(int low, int high, int toFind, int[] array) {
    this.low = low;
    this.high = high;
    this.toFind = toFind;
    this.array = array;
  }

  protected Boolean compute() {
    if (high - low <= 1) {
      return array[low] == toFind;
    }

    int middle = (low + high)/2;
    if (array[middle] > toFind) {
      BinSearch left = new BinSearch(low, middle, toFind, array);
      left.fork(); return left.join();  // Line X
    } else {
      BinSearch right = new BinSearch(middle, high, toFind, array);
      return right.compute();
    }
  }
}
```

For example,

```
int[] array = {1, 2, 3, 4, 6};
new BinSearch(0, array.length, 3, array).compute(); // return true
new BinSearch(0, array.length, 5, array).compute(); // return false
```

Assuming that we have a large number of parallel processors in the system and we never run into stack overflow, which of the following statement(s) about how `BinSearch` works is CORRECT?

(i) If we replace

```
left.fork(); return left.join();
```

on Line X with

```
return left.compute();
```

the search will likely run faster.

(ii) If we swap the order of `fork()` and `join()`, i.e., replace

```
left.fork(); return left.join()
```

with

```
left.join(); return left.fork();
```

the search will likely run faster.

(iii) Searching for the largest element in the input array will likely be faster than searching for the smallest element in the input array.

A. Only (i)

B. Only (ii)

C. Only (i) and (iii)

D. Only (ii) and (iii)

E. (i), (ii), and (iii)

Write `X` in the answer box if none of the combinations is correct.

Solution: C.

Note that `BinSearch` should not be parallelized at all since we always either search the left half or search the right half, never both at the same time.

In the given code, we could just call `left.compute()` instead of `left.fork(); return left.join();`. This reduces the overhead of interacting with the `ForkJoinPool` and therefore will likely be faster. (So (i) is correct).

(ii) is obviously wrong – calling `left.join()` before `left.fork()` would cause the task to block.

(iii) requires an understanding of the code in `compute()`. If the item `toFind` is smaller than the middle element, we search on the left. This means that the array is sorted in increasing order. So, searching for the smallest element would lead to the code keep going left (keep forking and joining); searching for the largest element would lead to the code keep going to the right (keep commputing). So, searching for the largest element is faster.

Questions 11 and 12 are based on the following asynchronous code:

```
CompletableFuture<A> cf1;
CompletableFuture<B> cf2;

cf1 = CompletableFuture.supplyAsync(() -> foo());
cf2 = cf1.thenApplyAsync(x -> bar(x));

B b = cf2.thenCombineAsync(
            cf1.thenApplyAsync(x -> thud(x)),
            (x, y) -> grunt(x, y))
         .join();
```

None of the methods in this question has wildcard / bounded wildcard as the type of the arguments. Every method invoked in the lambda expressions above is a pure function. The code compiles without any errors.

The relevant `CompletableFuture<T>` methods are (we removed the bounded wildcards for clarity):

`static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)`
Returns a new `CompletableFuture` that is asynchronously completed with the value obtained by calling the given `Supplier`.

`<U> CompletableFuture<U> thenApplyAsync(Function<T, U> fn)`
When the calling `CompletableFuture` completes normally, apply the given function `fn` to the result of the calling `CompletableFuture` asynchronously. It returns a new `CompletableFuture<U>` that encapsulates the result of `fn`.

`<U,V> CompletableFuture<V> thenCombineAsync(CompletableFuture<U> other, BiFunction<T, U, V> fn)`
When the calling `CompletableFuture` and `other` both complete normally, apply the given `BiFunction` to the results of these two `CompletableFuture` asynchronously. It returns a new `CompletableFuture<V>` that encapsulates the result of `fn`.

11. (3 points) Which of the following statements is CORRECT about the value of `b` after executing the above sequence?

    A. The value of `b` is equivalent to the value obtained by invoking
    ```
    b = grunt(bar(foo()), thud(foo()))
    ```
    synchronously.

    B. The value of `b` is equivalent to the value obtained by invoking
    ```
    b = grunt(thud(foo()), bar(foo()))
    ```
    synchronously.

    C. The value of `b` is equivalent to the value obtained by invoking
    ```
    b = grunt(thud(bar(foo()), foo()))
    ```
    synchronously.

    D. The value of `b` is undeterministic because the order of executing `bar` and `thud` is undeterministic.

    E. The variable `b` is uninitialized since `CompletableFutures` `cf1` and `cf2` are not joined.

    Write `X` in the answer box if none of the choices is correct.

> **Solution:** A.
>
> I think the only possible cause of confusion for this question is whether the answer should be A or B, but the type for the arguments to `thenCombinedAsync` should hint at A being the correct answer.

12. (3 points) Which of the following statement about the code above is INCORRECT?

    A. `bar` must return a value of type `B`.

    B. `grunt` must return a value of type `B`.

    C. The argument `x` to `thud` must have the type `B`.

    D. The first argument `x` to `grunt` must have the type `B`.

    E. The return type of `thud` is the same as the type of the second argument `y` to `grunt`.

    Write `X` in the answer box if none of the choices is correct.

> **Solution:** C. The argument to `thud` recieves the output from `foo` so it should be of type `A`.

Part II

Short Questions (34 points)

Answer all questions in the space provided on the answer sheet. Be succinct and write neatly.

13. (6 points) **Curry.**

The interface `TriFunction<S, T, U, R>` is a functional interface for a function that takes in three arguments, of types `S`, `T`, and `U` respectively, and returns a result of type `R`.

```
interface TriFunction<S,T,U,R> {
  R apply(S s, T t, U u);
}
```

Suppose we want to write a method `curry` that takes in a `TriFunction` and returns a curried version of the method.

```
.. curry(TriFunction<S, T, U, R> lambda) {
  // missing line
}
```

For instance, calling `curry` on

```
(x, y, z) -> x + y * z
```

returns

```
x -> y -> z -> x + y * z.
```

(a) (3 points) What should the return type of `curry` be?

Solution: `Function<S, Function<T, Function<U, R>>>`

(b) (3 points) Write the body of the method `curry`.

Solution: `return x -> y -> z -> lambda.apply(x, y, z);`

14. (3 points) **Compose.**

Write a method `compose` that returns a composition of a `Predicate<R>` with `Function<T,R>`. The returned function is a `Predicate<T>` that tests if the result of applying the given `Function<T,R>` matches the condition of the given `Predicate<R>`.

```
Predicate<T> compose(Function<T, R> f, Predicate<R> p) {
  // missing line
}
```

For example:

```
Predicate<String> predicate = compose(str -> str.length(), x -> x < 4);

Stream.of("I", "don't", "like", "green", "eggs", "and", "ham")
    .filter(predicate)
    .toArray();
```

returns the array with `["I", "and", "ham"]`.

Fill in the body of the method `compose`.

Recall that to evaluate a `Predicate p` on an input `x`, we call `p.test(x)`.

Solution: `return (x) -> p.test(f.apply(x));`

15. (6 points) **Substream.**

We say that a Stream `s` is a *substream* of another Stream `t`, if every element in `s` is contained in `t`. For example, a stream created with `Stream.of(0,0,1,2)` is a substream of the infinite stream created with `Stream.iterate(0, i -> i + 1)`.

The method `isSubstream` below returns `true` is `s` is a substream of `t` and `false` otherwise.

```
static <T> boolean isSubstream(Stream<T> s, Stream<T> t) {
 // missing line
}
```

(a) (3 points) Complete the body of the method `isSubstream`.

The following methods of `Stream<T>` are helpful (you may not need to use all):

`boolean allMatch(Predicate<? super T> predicate)`
Returns whether all elements of this stream match the provided predicate.

`boolean anyMatch(Predicate<? super T> predicate)`
Returns whether any elements of this stream match the provided predicate.

`boolean noneMatch(Predicate<? super T> predicate)`
Returns whether no elements of this stream match the provided predicate.

`Stream<T> filter(Predicate<? super T> predicate)`
Returns a stream consisting of the elements of this stream that match the given predicate.

`long count()`
Returns the count of elements in this stream.

Solution: This is a bad question, since a stream can only be scanned once. So, it cannot be solved in a single line using the APIs above.

The original intended answer, which many of you did write, is

`s.allMatch(x -> t.anyMatch(y -> y.equals(x)));`

This expression causes `IllegalStateException` if `s` has more than one items.

Everyone gets 3 free marks for this.

(b) (3 points) The method `isSubstream` above requires that both arguments must be streams containing the same type `T`. Suppose that we want to relax this constraint so that the type of `s` is a subtype of the type of `t`. Rewrite the method signature for `isSubstream` to make it so.

Solution: This question is, again, badly phrased :(

The question asks for the type of `t`, not the type of the elements in the stream `t`. So, possible answers are:

`isSubstream(Stream<T> s, Stream<? extends T> t)`

`isSubstream(Stream<T> s, Stream<?> t)`, etc

The first sentence, however, talks about the type of elements `T`. So many intepreted it as asking for the type of *elements* in `s` being a subtype of *elements* in `t`. In this case, possible answers are:

`isSubstream(Stream<T> s, Stream<? super T> t)` or

`isSubstream(Stream<? extends T> s, Stream<? super T> t)` or

`isSubstream(Stream<? extends T> s, Stream<T> t)`

I will accept answers match either one of the two intepretations.

Answers such as `isSubstream(Stream<? super T> s, Stream<? extends T> t)` that does not fit any of the two intepretation is still wrong.

16. (13 points) **Undoable.**

In this question, we are going to implement a class `Undoable`[1] that encapsulates a value that can be transformed with `flatMap` and can be restored to its previous state with the method `undo`.

Internally, an `Undoable<T>` object maintains a value of type `T` and a `LinkedList<Object>` storing a history of past values. When we apply a given function to the value, we also append the current value to the end of the `history` list. When we undo, we remove the last value from the `history` list, and replace the current value with this last value.

`Undoable<T>` is a monad – we can chain together different operations on an `Undoable` object using `flatMap`, and create a new `Undoable` object with an empty history using `of`.

For example, the following expression finds the length of the string "hello" and half its value:

```
Undoable<Double> d = Undoable.of("hello")
    .flatMap(s -> length(s))
    .flatMap(i -> half(i));
```

The `Undoable<Double>` object `d` now holds the value 2.5 and has a history list containing objects "hello" and 5, in that order.

Calling

```
Undoable<Integer> i = d.undo();
```

yields a new `Undoable<Integer>` object which holds the value 5 and has a history list containing object "hello". Calling

```
Undoable<String> s = i.undo();
```

gives a new `Undoable<String>` object which holds the value "hello" and has an empty history. Calling `s.undo()` would lead to an unchecked exception `CannotUndoException` being thrown.

Part of the class `Undoable` has been provided for you. We omit the `import` statements for brevity.

The following methods provided by `LinkedList<E>` could be useful:

`LinkedList()`
: Constructs an empty list.

`LinkedList(Collection<? extends E> c)`
: Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`boolean add(E e)`
: Appends the specified element to the end of this list.

`boolean addAll(Collection<? extends E> c)`
: Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

`E removeLast()`
: Removes and returns the last element from this list. Throws `NoSuchElementException` if this list is empty.

[1] UNDO-able, not UN-doable.

```
class CannotUndoException extends RuntimeException {
}

class Undoable<T> {
  T value;
  Deque<Object> history;

  Undoable(T t, Deque<Object> history) {
    this.value = t;
    this.history = history;
  }

  static <T> Undoable<T> of(T t) {
    return new Undoable<T>(t, new LinkedList<Object>());
  }

  public <R> Undoable<R> flatMap(Function<T, Undoable<R>> mapper) {
    // fill in the blank
  }

  public <R> Undoable<R> undo() {
    Deque<Object> newHistory = new LinkedList<>(this.history);
    R r;
    try {
      r = (R)newHistory.removeLast(); // Line A
    } catch (NoSuchElementException e) {
      // Missing line B
    }
    return new Undoable<R>(r, newHistory);
  }
}
```

(a) (3 points) Fill in the body of the method `length` below. This method takes in a `String str` and returns an `Undoable<Integer>` containing the length of the string `str`.

```
Undoable<Integer> length(String str) {
  :
}
```

Solution:

```
Undoable<Integer> length(String s) {
  Deque<Object> history;
  history = new LinkedList<>();
  history.add(s);
  return new Undoable<Integer>(s.length(), history);
}
```

Many of you simply wrote `return new Undoable<T>(str.length());`. You are not adding any history to the `Undoable` object, and as a result you won't be able to undo back to the String. You get 0 if your Undoable cannot undo.

Some of you add the length to the history instead of the string `str`. You get 1 mark if your Undoable can undo but return a wrong value.

(b) (3 points) Complete the body for method `flatMap`.

Solution:

```
public <R> Undoable<R> flatMap(Function<T, Undoable<R>> mapper) {
  Undoable<R> r = mapper.apply(value);
  Deque<Object> newHistory = new LinkedList<>();
  newHistory.addAll(history);
  newHistory.addAll(r.history);
  return new Undoable<R>(r.value, newHistory);
}
```

To get full marks, you need to do a few things right.

- retrieve the new value correctly with `mapper.apply()`.
- retrieve the new *history* correctly with `mapper.apply()`.
- combine the history by appending the new history `after` the current history.
- create a new `Undoable` with the new value and combined history.

Every error above gets you one mark off.

Most students append the current value or new value (instead of the new history) to the current history. If you append the value instead of the entire history from `mapper.apply()`, your `Undoable` is no longer a monad.

I got inspiration for this question after coming up with the `Logger/DoubleString` example in class. I thought this should be an easy question, given the parallel between the two (`Logger` keeps a history of what happend to a value, `Undoable` keeps a history of the values).

(c) (3 points) Explain why Line A would lead to a compiler warning of unchecked cast.

Solution: This is a narrowing type conversion, from R to Object. But since R is erased during compile time, the runtime system cannot safely check the type to make sure that it matches.

This is meant to be a difficult question, but it turns out to be too difficult – only a few students got this one correct.

Note that saying that it is a narrowing type conversion is not enough and is wrong – narrowing type conversion does not cause a compiler warning (you have seen this many times (e.g., in `equals`) so you should know this!).

Saying that the history keeps objects of different type and that the compiler warns us that we may assign variables of the wrong type is also not the correct answer. If we do not use a generic type here, there is actually no problem (e.g., as you have seen in `equals` and exercises related to type conversion).

(d) (3 points) Let's say that we leave Line A as it is and ignore the compilation warning. What would happen if we do the following? Explain.

```
Undoable<Integer> i = Undoable.of("hello").flatMap(s -> length(s));
Undoable<Double> d = i.undo();
```

Solution: The code runs without error. Even though we assign an Undoable<String> to Undoable<Double>, during runtime, it is stored as an Object reference, and the reference can refer to String. An error would occur only if we try to apply function that operates on Double to the Undoable, in which case it will throw a ClassCastException.

This is another difficult question to test if you understand type erasure. Again, not many students got this one right. Most of you thought that a ClassCastException will be thrown.

(e) (1 point) The missing Line B should throw a `CannotUndoException`. Fill in this missing line.

Solution: `throws new CannotUndoException();`

17. (6 points) **LazyList.**

The class `LazyList` defines a possibly infinite list using lazy evaluation. It is a simpler version of `InfiniteList` you see in class – it does not cache the computed head / tail and supports only `iterate`, `empty`, `isEmpty`, `map`, `concat`, and `forEach` operation. The definition of `empty` and `isEmpty` methods are ommited for brevity. The `iterate` method is slightly different, as it supports a condition to stop iterating.

```
class LazyList<T> {
  private Supplier<T> head;
  private Supplier<LazyList<T>> tail;

  public LazyList(Supplier<T> head, Supplier<LazyList<T>> tail) {
    this.head = head;
    this.tail = tail;
  }

  public static <T> LazyList<T> iterate(T init, Predicate<T> cond,
      UnaryOperator<T> op) {
    if (!cond.test(init)) {
      return LazyList.empty();
    } else {
      return new LazyList<T>(
          () -> init,
          () -> iterate(op.apply(init), cond, op));
    }
  }

  public <R> LazyList<R> map(Function<T, R> mapper) {
    return new LazyList<R>(
        () -> mapper.apply(head.get()),
        () -> tail.get().map(mapper));
  }

  public T forEach(Consumer<T> consumer) {
    LazyList<T> list = this;
    while (!list.isEmpty()) {
      cosumer.accept(list.head.get());
      list = list.tail.get();
    }
  }

   :
}
```

(a) (3 points) Suppose we call

`LazyList.iterate(0, i -> i < 2, i -> i + 1).map(f).map(g).forEach(c)`

where `f` and `g` are lambda expressions of type `Function` and `c` is a lambda expression of type `Consumer`. Let `e` be the lambda expression `i -> i + 1` passed to `iterate`.

Write down the sequence of which the lambda expressions `e`, `f`, `g`, and `c` are evaluated.

> **Solution:** fgce fgce
>
> If you made careless mistakes and answered "efgcefgc" or only "fgce" you still get 2 marks (which most of you did).
>
> There are multiple questions during the exam, about whether I am asking for the sequence when the lambda expressions start their evaluation, or end their evaluation. There is actually no difference! Note that one lambda expression does not call another lambda expression.

(b) (3 points) The method `concat` takes in two `LazyList` objects, `l1` and `l2`, and creates a new `LazyList` whose elements are all the elements of the first list `l1` followed by all the elements of the second list `l2`. The elements in newly concatenated list must be lazily evaluated as well. For example, in

```
LazyList<Integer> l1 = LazyList.iterate(0, i -> i < 2, i -> i + 1);
LazyList<Integer> l2 = LazyList.iterate(5, i -> i < 8, i -> i + 2);
LazyList<Integer> l3 = LazyList.concat(l1, l2);
l3.forEach(x -> System.out.print(x));
```

Note that, being a lazily evaluated list, nothing is evaluated when `l3` is created. Thus, `concat` should not result in an infinite loop if the list `l1` infinitely long. The elements are only evaluated when terminal operator `forEach` is called. In the example above, `0157` will be printed.

Complete the body of the method `concat`.

> **Solution:**
>
> ```
> public static <T> LazyList<T> concat(LazyList<T> l1, LazyList<T> l2) {
>   if (l1.isEmpty()) {
>     return l2;
>   } else {
>     return new LazyList<T>(l1.head, () -> concat(l1.tail.get(), l2));
>   }
> }
> ```
>
> This should be an easy question based on Lab 3 but yet many of you wrote code that are pretty far away from the solution. For those solutions that are close, some common mistakes are (i) not considering empty list (-1 marks); (ii) concat in the wrong order (`concat(l2, l1.tail.get()` or `concat(l1, l2.tail())`) (-2 marks).
>
> Many of you also like to write `new LazyList<>(l2.head, l2.tail)` – which is just `l2`! Another popular long-winded expression is `() -> l1.head.get()`, which is just `l1.head`.

END OF PAPER