NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 1 AY2017/2018

CS2030 Programming Methodology II

November 2017 Time Allowed 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 17 questions and comprises 10 printed pages, including this page.
2. Write all your answers in the answer sheet.
3. The total marks for this assessment is 80. 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, select the most appropriate answer and **write your answer in the corresponding answer box on the answer sheet.** Each question is worth 3 points.
- If multiple answers are equally appropriate, pick one and write the chosen answer in the answer box. Do NOT write more than one answer in the answer box.
- If none of the answers are appropriate, write X in the answer box.

1. (3 points) Which of the following is NOT a principle of object-oriented programming?
   A. Abstraction
   B. Encapsulation
   C. Immutability
   D. Inheritance
   E. Polymorphism

   Write X in the answer box if none of the answer above are correct.

> **Solution:** C.
>
> This is a give-away question.

2. (3 points) Which of the following statements about good OO programming practice is LEAST appropriate?

    A. We should use inheritance to reduce duplication of code between two classes as much as possible.

    B. We should declare fields as `private` as much as possible.

    C. We should not throw exceptions that reveal internal implementation of a class as much as possible.

    D. We should avoid using accessors and mutators (also known as getters and setters) to private fields as much as possible.

    E. We should use polymorphism so that each class is responsible for handling its own behavior as much as possible.

    Write `X` in the answer box if none of the answers above are correct.

Solution: A.

Inheritance is not always the right solution to reduce code duplicates. We should follow LSP, and use composition when appropriate.

3. (3 points) Late binding in Java implies that:
   A. Evaluation of an expression is delayed until it is needed.
   B. The method to invoke is known only during runtime, not during compile time.
   C. Exception are caught and processed only when the program exits, not immediately after the exception is thrown.
   D. Asynchronous tasks are delayed until the method `bind` is called.
   E. The value stored in a `CompletableFuture` object is known only after the object completes.

   Write `X` in the answer box if none of the answers above are correct.

Solution: B.

Nothing much to say here. Should be a straight forward question.

4. (3 points) Suppose we have the following interface and classes:

```
interface I {
  void f();
}

abstract class A implements I {
  abstract void g();
  abstract void g(int x);
  void h() {
  }
}

class B extends A {
  :
}
```

Consider the methods involved:

(i) `f()`
(ii) `g()`
(iii) `g(int x)`
(iv) `h()`

In order to be able to create an instance of `B`, it is sufficient to implement the following in class `B`:

A. Both (ii) and (iii)
B. Either one of (ii) and (iii)
C. (i), and either one of (ii) and (iii)
D. (ii), (iii) and (iv)
E. (i), (ii), (iii), (iv)

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

Solution: I received many questions about this question and realized during the exam that some students might not be familiar with the terms *sufficient* and *necessary* (Taught in CS1231)

What the question is asking is that, what to implement to guaratee that we can create an instance of `B`. We only need to implement (i), (ii), and (iii). We do not need to implement (iv).

If you interpret *sufficient* in the context of logical inference, then `E` is the right answer. Implementing all (i)-(iv) implies that we can create a new instance of `B`.

If you interpret *sufficient* as *just enough*, then (i)-(iii) is the answer, and you might choose `X`.

I accept both `E` and `X` as correct answer due to the phrasing of this question.

5. (3 points) Suppose we have a variable `i` of type `int` that has been initialized elsewhere. We assign `i` to two `Integer` objects as follows:

```
Integer x = i;
Integer y = i;
```

Select the most appropriate statement about the code snippet above.

A. `x == y` always evaluates to `true`

B. `x == y` always evaluates `false`

C. `x == y` may evaluate `true` or `false`

D. `x.equals(y)` may evaluate `true` or `false`

E. `x.equals(y)` always evaluates `false`

Write `X` in the answer box if none of the answers above are correct.

Solution: C.

Depending on the value of `i` and whether Java cached the object internally, `x == y` can be either true or false. `x.equals(y)` is always true.

6. (3 points) Suppose we have a generic class `E` and we want the type parameter to `E` to be a subclass of `E`. In other words, we allow `E<F>` only if class `F` is either `E` or inherits from `E`. Which of the following declarations of `E` ensures this?

   A. `class E<T> implements T`

   B. `class E<T> extends E`

   C. `class E<T> extends E<T>`

   D. `class E<T extends E>`

   E. `class E<T extends E<T>>`

   Write `X` in the answer box if none of the answers are correct.

Solution: Both D and E.

This is just like `enum`. Again, straight out of the notes. For D, there will be a compiler warning that `E` is a raw type, but still works.

7. (3 points) Consider the functions below:

```
int fff(int i) {
  if (i < 0) {
    throw new IllegalArgumentException();
  } else {
    return i + 1;
}

int ggg(int i) {
  System.out.println(i);
  return i + 1;
}

int hhh(int i) {
  return new Random().nextInt() + i;
}

int jjj(int i) {
  return Math.abs(i);
}
```

Which of the above is a pure function?

A. Only `fff`

B. Only `jjj`

C. Only `fff` and `ggg`

D. Only `ggg` and `hhh`

E. Only `fff` and `jjj`

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

Solution: B.

Almost everyone got this one correct. A pure function cannot have side effects – cannot throw exceptions (not `fff`, cannot perform I/O (not `ggg`), and must always return the same output given the same input (not `hhh`).

8. (3 points) Consider a generic class `A<T>` with a type parameter `T` and a constructor with no argument. Which of the following statements are valid (i.e., no compilation error) ways of creating a new object of type `A`? We still consider the statement as valid if the Java compiler produces a warning.

   (i) `new A<int>();`

   (ii) `new A<>();`

   (iii) `new A();`

   A. Only (i)

   B. Only (ii)

   C. Only (i) and (ii)

   D. Only (ii) and (iii)

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

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

Solution: D.

A generic type cannot be primitive type, so we know (i) would lead to an error.

(ii) and (iii) are OK – for both, Java will create a new class replacing `T` with `Object`.

9. (3 points) Which of the following statement(s) about the following Java program is/are TRUE?

```
class A {
  int field;
  void method() {
    Function<Integer, Integer> func = x -> field + x;
  }
}
```

(i) `field` is stored on the heap

(ii) `func` is stored on the stack.

(iii) `func` refers to an object stored on the heap.

(iv) `x` is not stored anywhere.

A. Only (i)

B. Only (i) and (ii)

C. Only (ii) and (iii)

D. Only (i), (ii) and (iv)

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

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

Solution: When I set this question, I didn't mean to ask where the variables are stored at a specific time. I am just asking, just generally, where would they be stored as the program runs? `A` is a class so the instance field `field` would be on the heap. `func` is a local varialble, so it would go on the stack. `func` refers to the lambda expression, which is internally implemented as an anonymous class, so it refers to something on the heap. Finally, `x` is an argument to the lambda expression, so it is not stored anywhere. My intended answer is `E`.

This question triggered many questions during exam, that I decided to change the question to add: where would they be stored *after we execute* `A a = new A(); a.method();`.

With this addition, the question become much easier than I intended. After the two lines are code are executed, only `a` (and `field`) remains on the heap. The answer is `X` since only (i) and (iv) are correct.

10. (3 points) Consider the program below. The method `doSomething()` may run for an undeterministic amount of time.

```
import java.util.concurrent.CompletableFuture;

class CF {
  static CompletableFuture<Void> printAsync(int i) {
    return CompletableFuture.runAsync(() -> {
        doSomething();
        System.out.print(i);
    });
  }

  public static void main(String[] args) {
    printAsync(1);
    CompletableFuture.allOf(printAsync(2), printAsync(3)).join();
  }
}
```

Which of the following are possible output printed by the program if `main` runs to completion normally?

(i) `123`

(ii) `213`

(iii) `1`

(iv) `32`

A. Only (i)

B. Only (i) and (ii)

C. Only (ii)

D. Only (i), (ii) and (iv)

E. Only (iii) and (iv)

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

Solution: D.

`printAsync(1)` is not synchronized so it can get printed anytime, or not printed at all before the program exits. But, there is a call to `join()` for `printAsync(2)` and `printAsync(3)`. So, we know for sure that both `2` and `3` will be printed (in any order).

11. (3 points) Consider the following incomplete code to compute Fibonacci number as a `RecursiveTask`. We want to complete the method `compute` by inserting one more line of code. Which of the following lines of code, when inserted, would compile without error and would lead to the *correct* Fibonacci number for 10 being returned when `ForkJoinPool.commonPool().invoke(new Fib(10))` is called?

```
class Fib extends RecursiveTask<Integer> {
   final int n;
   Fib(int n) { this.n = n; }
   Integer compute() {
     if (n <= 1)
        return n;
     Fib f1 = new Fib(n - 1);
     Fib f2 = new Fib(n - 2);
     // insert code here
   }
 }
```

A. `return f1.compute() + f2.compute();`

B. `return f1.join() + f2.join();`

C. `return f1.fork() + f2.fork();`

D. `return f1.compute() + f2.fork();`

E. `return f1.fork() + f2.join();`

Solution: A.

If we invoke `fork`, we need to `join` to get the result back. None of the options that uses `fork` also `join` back the result. The only option that gives us the correct result is `A`. Note that it computes the Fibonacci number sequentially.

12. (3 points) Consider the same incomplete code to compute Fibonacci number in the previous question. Again, we want to complete the method `compute` by inserting one line of code. Which of the following lines of code, when inserted, would compile without error and lead to *correct* and *efficient* parallelization of the calculation of Fibonacci number for 10 when `ForkJoinPool.commonPool().invoke(new Fib(10))` is called?

```
class Fib extends RecursiveTask<Integer> {
   final int n;
   Fib(int n) { this.n = n; }
   Integer compute() {
     if (n <= 1)
        return n;
     Fib f1 = new Fib(n - 1);
     Fib f2 = new Fib(n - 2);
     // insert code here
   }
 }
```

A. `f1.fork(); f2.fork(); return f1.join() + f2.join();`

B. `f1.fork(); f2.fork(); return f2.join() + f1.join();`

C. `f1.fork(); return f1.join() + f2.compute();`

D. `f1.fork(); return f2.compute() + f1.join();`

E. `return f1.compute() + f2.compute();`

Solution: D.

All the choices give correct answers (actually, it hints at the answer for the previous question :)) E computes sequentially, so is the least efficient. A and B are the same – they gives up the current thread and launch two new tasks. C actually runs sequentially as well – it makes sure f1 is completed, before running f2. D is the most efficient (slightly better than A and B).

Part II

Short Questions (44 points)

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

13. (3 points) **Predicate.** The method `and` below takes in two `Predicate` objects `p1` and `p2` and returns a new `Predicate` that evaluates to `true` if and only if both `p1` and `p2` evaluate to `true`.

```
Predicate<T> and(Predicate<T> p1, Predicate<T> p2) {
  return ...
}
```

Fill in the body of the method `and`.

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

Solution: The simplest answer is:

```
return x -> p1.test(x) && p2.test(x);
```

A couple of you explicitly create the anonymous class, which is also correct:

```
return new Predicate<T>() {
  public boolean test(T x) {
    return p1.test(x) && p2.test(x);
  }
}
```

Many of you did something in between:

```
return new Predicate<T>( x -> p1.test(x) && p2.test(x); )
```

which is not correct and received a -1 penalty.

Some of you wrote :

```
return p1.test(x) && p2.test(x);
```

which returns a `boolean` and eagerly evalutes the predicates. You get 1 mark.

14. (3 points) **SAM.** The interface `SummaryStrategy` has a single abstract method `summarize`, allowing any implementing class to define its own strategy of summarizing a long `String` to within the length of a given `lengthLimit`. The declaration of the interface is as follows:

```
@FunctionalInterface
interface SummaryStrategy {
  String summarize(String text, int lengthLimit);
}
```

There is another method `createSnippet` that takes in a `SummaryStrategy` object as an argument.

```
void createSnippet(SummaryStrategy strategy) { ... }
```

Suppose that there is a class `TextShortener` with a static method `String shorten(String s, int n)` that shortens the String `s` to within the length of `n`. This method can serve as a summary strategy, and you want to use `shorten` as a `SummaryStrategy` in the method `createSnippet`.

Show how you would call `createSnippet` with the static method `shorten` from the class `TextShortener` as the strategy.

Solution:
```
createSnippet((s, n) -> TextShortener.shorten(s, n));
```
or
```
createSnippet(TextShortener::shorten);
```

15. (4 points) **Stream.**

(a) (1 point) What is the value of the variable x after executing the following statement?

```
Stream.of(1,2,3,4)
    .reduce(0, (result, x) -> result * 2 + x);
```

Solution: 26

(b) (3 points) After we parallelized the above code into:

```
Stream.of(1,2,3,4)
    .parallel()
    .reduce(0, (result, x) -> result * 2 + x);
```

We found that the output is different. Why?

Solution: `2*result + x` is not associative.

The above is sufficient to get you 3 marks. Some of you explained that the ordering is different when you reduce is parallel. You get 2 marks. Since reducing in parallel and in different order can still get you the right answer as long as the reduction operation is associative. To get 3 marks, you have to additionally say that the result depends on the order of reduction (basically a long winded way to say the function is non-associative).

An answer that says, the output is different because the reduction runs in parallel, will get 0 mark. This answer simply repeats the question (why is the output different when we run in parallel?)!!

Answers that say the reduce method has side effect, or is stateful, or does not have a combiner, are not correct. These answers, even if they are correct, apply also to

```
Stream.of(1,2,3,4).reduce(0, (result, x) -> result + x);
```

But the code above can be parallized!

16. (18 points) Lazy.

In this question, you are going to implement a class `LazyInt` that encapsulates a lazily evaluated `Integer` value. A `LazyInt` is specified by a `Supplier`. When the value of the `LazyInt` is needed, the `Supplier` will be evaluated to yield the value. Otherwise, the evaluation is delayed as much as possible. `LazyInt` supports `map`, `flatMap`, and `get` operations:

- `map` returns a `LazyInt` consisting of the results of applying the given function to the value of this `LazyInt`.
- `flatMap` returns a `LazyInt` consisting of the results of replacing the value of this `LazyInt` with the value of a mapped `LazyInt` produced by applying the provided mapping function to the value.
- `get` returns the value of `LazyInt`.

For example, the expression

```
new LazyInt(() -> 10).map(x -> x * x).flatMap(x -> new LazyInt(() -> x * 2)).get()
```

will return 200.

Part of the class `LazyInt` has been provided for you. We omit the `import` statements for brevity. Recall that, to evaluate a lambda expression of type `Function`, we need to invoke the `apply` method of `Function`.

```
class LazyInt {
  Supplier<Integer> supplier;

  LazyInt(Supplier<Integer> supplier) {
    this.supplier = supplier;
  }

  int get() {
    return supplier.get();
  }

  LazyInt map(Function<? super Integer, Integer> mapper) {
    // TODO
  }

  LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
    // TODO
  }
}
```

(a) (2 points) Complete the body for the method `map`.

Solution: `return new LazyInt(() -> mapper.apply(get()));`
Most of you get this correct.

(b) (4 points) Complete the body for the method `flatMap`.

Solution: The answer follows from the previous question:

```
return new LazyInt(() -> mapper.apply(get()).get();
```

Surprisingly, this question stumped many of you.

A common answer is to return `mapper.apply(get())`, but this forces the evaluation of `mapper` and `supplier`, and is no longer lazy. Another commonly provided answer is: `return new LazyInt(mapper.apply(get()).supplier);`, which is again not lazy. You get only two marks for being right but not being lazy.

Other common wrong answers (0 marks) with incompatible type:

- `return mapper.apply(this);`
- `return mapper.apply(get()).apply(get());`
- `return new LazyInt(() -> new LazyInt(() -> mapper.apply(get()).get()));`
- Variations that use `map` to implement `flatMap`, using `map(mapper)`

(c) (4 points) Is `LazyInt` a functor? Explain.

Solution: Yes. Calling `map` with `identity` function gives a `LazyInt` of

```
identity.apply(supplier.get())
```

which is just `supplier.get()` – no change.

Calling `map(g).map(f)` gives a `LazyInt` of

```
f.apply(new LazyInt(() -> g.apply(get())).get()),
```

which is just `f.apply(g.apply(get()))`.

I expected students to explain *why* the Functor laws hold. Except a few of students, however, most simply answered that LazyInt follows the identity law and composition law, then recites what the laws are, without explanation of why LazyInt follows the law. I decided to be lenient:

- if you explain what the laws are and linked it to the context of LazyInt, I still give you full marks.
- if you just recite the name of the laws, you get 2 marks.
- if you just say "`LazyInt` follows the functor laws", you get 1 mark.

Answers that only mention `LazyInt` matches the interface of `Functor` and supports `map` will not get any marks. Not all classes that support `map` is a functor.

(d) (4 points) Is `LazyInt` a monad? Explain.

Solution:

There are several possible answers.

If you say, no, because it does not have a `of` (or unit) operator, I will give you full marks.

If you assume that `new LazyInt(() -> x)` is the `of` operator for value `x`, and explain why

the `LazyInt` follows that law of monads, you get full marks (there is probably not enough space to write it all down!) So,

- if you explain what the laws are and linked it to the context of LazyInt, I still give you full marks.
- if you just recite the name of the laws, you get 2 marks.
- if you just say "`LazyInt` follows the monad laws", you get 1 mark.

Answers that only mention `LazyInt` matches the interface of `Monad` and supports `flatMap` will not get any marks. Not all classes that support `flatMap` is a monad.

(e) (4 points) Why is it better to declare the argument to `map` as `Function<? super Integer, Integer>` instead of `Function<Integer, Integer>`? You may want to give an example.

Solution: This is designed to be an easy question. Surprisingly, despite students encountering something similar to `Function<? super Integer, Integer>` many times, many cannot answer this question.

Students who answer something unrelated or mixed up `super` and `extends` in bounded generic types will get 0 mark.

Some students only explained the difference between `? super Integer` and `Integer`, but did not explain why it is better. I give 1 mark.

To get full marks, you have to explain *why it is better*. The argument is declared this way so that we can pass in any function that operates on the superclass of `Integer` (such as `Number` and `Object`) (e.g., `Function<Object,Integer> h = x -> x.hashCode()`) into map.

17. (16 points) Collection

A sparse matrix is a matrix where most of the entries are zeros. As such, it is inefficient in terms of memory to store the elements of the matrix with a 2D array. In this question, we will explore alternative ways to store the matrix elements. We only need to store non-zero elements. We can define a matrix element as follows:

```
class Element {
  public int row;
  public int col;
  public double value;

  @Override
  public boolean equals(Object o) { .. }

  @Override
  public int hashCode() { .. }
}
```

You can assume that the methods `equals` and `hashCode` have been implemented.

Our matrix implementation needs to support the two operations:

- `double get(int row, int col)`: return the content of the matrix at the given `row` and `col`.
- `void set(int row, int col, int value)`: set the value of the matrix at the given row and col to `value`.

One way to implement the class `Matrix` is to use `HashMap`. Recall that `HashMap` has two generic type parameters `K` (for the key) and `V` (for the value). The two most relevant methods for `HashMap` is `public V get(Object key)`, which returns the value to which the specified key is mapped, or null if this map contains no mapping for the key, and `public V put(K key, V value)`, which associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

(a) (4 points) Show the line of code that declares a field of type `HashMap` in the class `Matrix` to store the `Element` objects. Explain why you choose to design your `HashMap` this way (in particular, your choice of keys and values).

Solution: `HashMap<Pair<Integer,Integer>,Element> map = new HashMap<>();`
Using both row and col as key (as a Pair) allow us to look up efficiently a given row and col, as long as `Pair` computes its hashCode using both the row and col value.
Other acceptable answers include devising your own hash function that maps row and col to a unique integer or string, then use `HashMap<Integer,Element>` or `HashMap<String,Element>`.
Another possible answer is `HashMap<Integer, HashMap<Integer,Element>>`. The first HashMap hashes using row, to another HashMap using cols. We need to hash twice, but the advantage of this is that we can easily retrieve elements from the same row.

(b) (4 points) Using your declaration above, explain how you would implement the method `get`.

Solution: Call `map.get(new Pair(row, col))`. If result is null, return 0. Otherwise, return the value.

Other possible answers are possible, depends on (a). But many of you did not consider what happened when `get()` returns `null` (the method should return 0). If you forget this, you get 2 out of 4 marks.

A better answer also mentions checking if row and col are in range of the matrix.

(c) (4 points) Using your declaration above, explain how you would implement the method `set`.

Solution: Basically you need something like `map.put(new Pair(row, col), value)`. A better answer handles what happen when value is 0 – if there is an existing entry, remove it; Otherwise, we do nothing (no point inserting 0 into our sparse matrix).

A better answer also mentions checking if row and col are in range of the matrix.

(d) (4 points) Should `Element` be declared as an inner class within `Matrix`? Why or why not?

Solution: Yes. The class `Element` is only used within the `Matrix` class.

If you said no, you have to give a good reason (for instance a situation where Elements are needed outside of Matrix).

END OF PAPER