NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 2 AY2020/2021

CS2030S Programming Methodology II

April 2021 Time Allowed 120 minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 25 questions and comprises 21 printed pages, including this page.
2. The total marks for this assessment is 80. Answer **ALL** questions.
3. This is a **OPEN BOOK** assessment. You are only allowed to refer to hardcopies materials.
4. All questions in this assessment paper use Java 11.

Types (5 marks)

1. Consider the following code:

```
Integer i = 0;
Object o = (Number) i;
```

Which of the following statement(s), if any, is true?

A. The runtime type of `o` is `Integer`.

B. The compile-time type of `i` is `Object`.

C. `i`'s compile-time type is inferred to be `Number`.

D. The type of `i` has been erased to `Object`.

E. The code would cause a compilation error.

F. The code would cause a run-time error.

Solution: This question assesses the understanding of basic concepts related to types.

The compile-time type of `i` is `Integer`; for `o` is Object. Since `i` points to 0, `o` will also points to 0. And so, `o` has the runtime type of `Integer`. Type inference is not involved since all the types are spelled out. Type erasure is not involved since generics are not used. The code would compile fine since we have a widening type conversion from `Number` to `Object`.

Only A is true.

Overriding (7 marks)

2. Consider the following definition of the classes `Parent` and `ParentException`:

```
class ParentException extends Exception {
}

class Parent<T> {
  public <R> Parent<R> foo(Parent<? extends T> p) throws ParentException {
    :
  }

  // insert class Child here
}
```

Which of the following definition(s), if any, of the nested class `Child`, when inserted into the class above, will successfully override the method `foo` in `Parent` without compilation warning or error?

A.
```
// No @Override
private class Child<S> extends Parent<S> {
  public <R> Parent<R> foo(Parent<? extends S> p) throws ParentException {
    return null;
  }
}
```

Solution: OK. It is OK to leave out `@Override`, which is just an annotation.

B.
```
// Different exception thrown
private class Child<S> extends Parent<S> {
  @Override
  public <R> Parent<R> foo(Parent<? extends S> p) throws Exception {
    return null;
  }
}
```

Solution: Error. Since the child method is trying to throw a more general exception. This behavior violates LSP and the Java compiler does not allow it.

C.
```
// Different access modifier
private class Child<S> extends Parent<S> {
  @Override
  private <R> Parent<R> foo(Parent<? extends S> p) throws ParentException {
    return null;
  }
}
```

Solution: Error. Since the child method is now private, and can't be accessed. This behavior violates LSP and the Java compiler does not allow it.

D.
```
// No exception thrown
private class Child<S> extends Parent<S> {
  @Override
```

```
    public <R> Parent<R> foo(Parent<? extends S> p) {
      return null;
    }
  }
```

Solution: OK. A `Child<S>` instance that does not throw an exception can substitute a `Parent`. No LSP is violated.

E.
```
// Different return type
private class Child<S> extends Parent<S> {
  @Override
  public <R> Child<R> foo(Parent<? extends S> p) throws ParentException {
    return null;
  }
}
```

Solution: OK. We can return a more specific type (i.e., a subtype) when we override. We discussed this in length in our recitation.

F.
```
// Different parameter type
private class Child<S> extends Parent<S> {
  @Override
  public <R> Parent<R> foo(Parent<S> p) throws ParentException {
    return null;
  }
}
```

Solution: Error. `Parent<S>` and `Parent<? extends S>` are considered two different types at compile time, despite both being erased to `Parent` after erasure.

G.
```
// Different type variable
private class Child<S> extends Parent<S> {
  @Override
  public <T> Parent<T> foo(Parent<? extends S> p) throws ParentException {
    return null;
  }
}
```

Solution: OK. We are just using a different name for the type parameter. The new parameter `T` shadows the `T` declared in `Parent<T>`, but that is OK.

LSP (5 marks)

3.
```
public class Card {
    public final static int SPADES = 0;
    public final static int HEARTS = 1;
    public final static int DIAMONDS = 2;
    public final static int CLUBS = 3;
    private final int suit;
    private final int cardNumber;

    public Card(int suit, int cardNumber) {
        this.suit = suit;
        this.cardNumber = cardNumber;
    }

    public int getCardNumber() {
        return this.cardNumber;
    }
}

public class CardDeck {
    private final static int cardTotal = 52;

    public CardDeck() {
        // Implementation removed
        // Creates a card deck
    }

    /**
    * Returns the number of cards in the deck that have a card number less than n.
    *
    * @param cards  an array containing a deck of cards
    * @param n the card number
    * @return the number of cards in deck with a card number less than n
    */
    public static int countCardsLessThanN(Card[] cards, int n) {
        int count = 0;
        for (int i = 0; i < cardTotal; i++) {
            if (cards[i].getCardNumber() < n) {
                count++;
            }
        }
        return count;
    }
}
```

Each card has a suit and a card number, for example, to create the two of spades you would use `new Card(Card.SPADES, 2);`

If you wanted to create the king of hearts you would use `new Card(Card.HEARTS, 13);`

In order to handle the face cards (Ace, Jack, Queen, and King), someone decides to implement a new `FaceCard` class, which inherits from `Card`.

```
public class FaceCard extends Card {
    public final static int ACE = 0;
    public final static int JACK = 1;
    public final static int QUEEN = 2;
```

```
    public final static int KING = 3;
    private int faceType;

    public FaceCard(int suit, int faceType) {
        super(suit, -1);
        this.faceType = faceType;
    }
}
```

`FaceCard` violates the Liskov Substitution Principle. True or false? Explain your reasoning below.

Solution: Yes, it violates LSP. If we put `FaceCard` into the array `cards` and pass it into `countCardsLessThanN`, the function would return a wrong answer, since a `FaceCard` always have `cardNumber` initialized to -1 and will always be counted by `countCardsLessThanN`.

Overloading (3 marks)

4. Consider the following class `ArraySort`. This is a class that contains many sort methods to allow you to `sort` different types of `Array`s.

```
public class ArraySort {

    public int sort(Array<String> arrayString) {
        // Implementation omitted
    }

    public int sort(Array<Integer> arrayInteger) {
        // Implementation omitted
    }
}
```

This code will compile without warning or error. True or false?

Explaining your reasoning below.

Solution: False. We cannot overload two methods with the same method signature (after type erasure).

Stack and Heap (4 marks)

Consider the following classes:

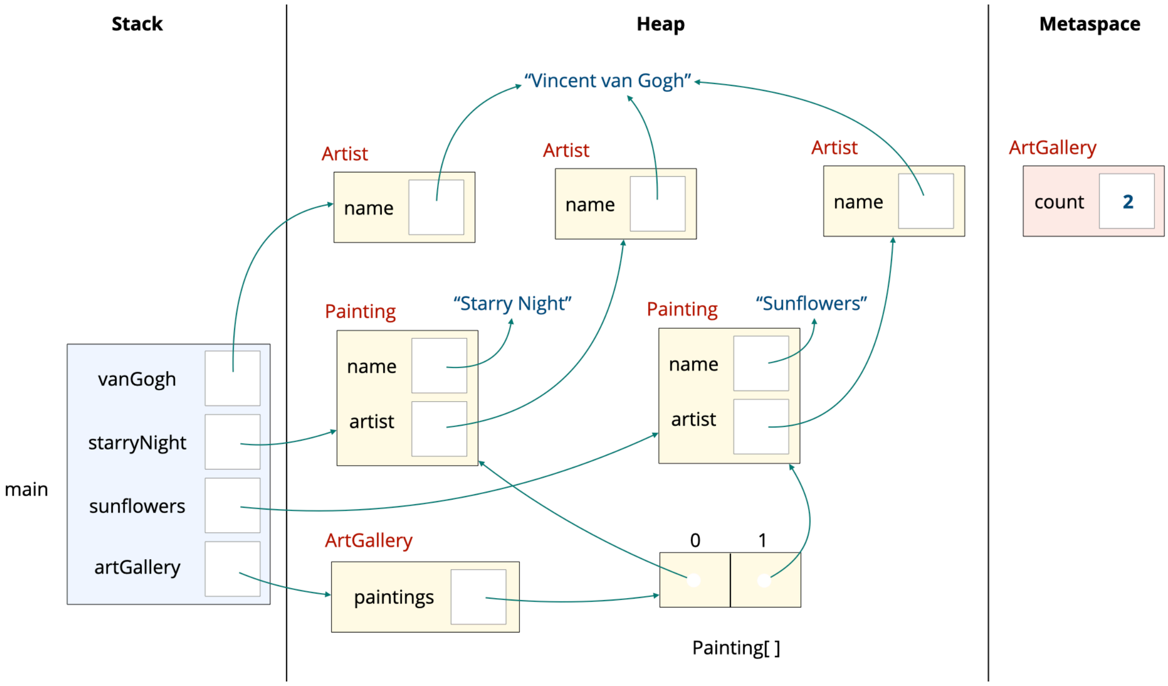
```
class ArtGallery {
    // Line A  private static int count = 0;
    private Painting[] paintings = new Painting[2];

    public void addPainting(Painting painting) {
        paintings[ArtGallery.count] = painting;
        // Line B  this.count++;
    }
}

class Artist {
    private String name;

    public Artist(String name) {
        this.name = name;
    }

        public String getName() {
          return name;
        }
}

class Painting {
    private String name;
    private Artist artist;

    public Painting(String name, Artist artist) {
        this.name = name;
        // Line C  this.artist = new Artist(artist.getName());
    }
}
```

as looking at the diagram, each painting points to a new instance of an Artist

and the following `main` method:

```
public static void main(String[] args) {
        Artist vanGogh = new Artist("Vincent van Gogh");
        Painting starryNight = new Painting("Starry Night", vanGogh);
        Painting sunflowers = new Painting("Sunflowers", vanGogh);
        ArtGallery artGallery = new ArtGallery();
        // Line D  artGallery.addPainting(starryNight);
        // Line E  artGallery.addPainting(sunflowers);
}
```

Consider the following diagram showing the stack, heap, and metaspace just after the end of Line E. Note that (i) we simplified the content of metaspace and representation of strings in the diagram for brevity, (ii) the items in the stack frame are from top to bottom, and (iii) `args` is not included in the stack frame.



5. Complete Line A in the given code so that the stack/heap/metaspace corresponds to what is shown above.

Solution: `private static int count = 0;`

6. Complete Line B in the given code so that the stack/heap/metaspace corresponds to what is shown above.

Solution: `count += 1;`

7. Complete Line C in the given code so that the stack/heap/metaspace corresponds to what is shown above.

Solution: `this.artist = new Artist(artist.getName());`

8. Complete Lines D and E in the given code so that the stack/heap/metaspace corresponds to what is shown above.

Solution:

```
artGallery.addPainting(starryNight);
artGallery.addPainting(sunflowers);
```

Exception (6 marks)

You are re-writing a program for the Marina Bay Sands hotel that was written by a student who has not taken CS2030S! This program helps them check if guests are currently staying in the hotel or if they have already checked out. You have encountered a method called `checkGuestStatus`, which will return `false` if a guest is already checked out, and `true` if the guest is still staying. This program includes a new type of exception called `NoSuchGuestException`, which directly inherits from `RuntimeException`. In addition, you have a method called `lookUpGuest`, which will return the guest details and return them if they are present. Otherwise, the `lookUpGuest` method throws the `NoSuchGuestException` exception.

```java
public boolean checkGuestStatus(String name) {
    try {
        lookUpGuest(name);
        return true;
    } catch (NoSuchGuestException e) {
        return false;
    } catch (Exception e) {
        return false;
    }
}
```

9. With regards to exceptions in Java and the code above, which of the following statements are true?

   A. A method that might throw an unchecked exception requires a try-catch block around its invocation.

   B. Checked exceptions are used for errors that the programmer cannot foresee.

   C. In Java, exceptions are primitives.

   D. `NoSuchGuestException` is an unchecked exception.

> **Solution:** Unchecked exceptions do not require a try-catch block. Checked exceptions do – so the programmer anticipates (foresee) that a checked exception will happen. Exceptions are reference types.
>
> The answer is D. All exceptions inherit from `RuntimeException` are unchecked exceptions.

10. There are several design problems with this program, the `checkGuestStatus` method, and the `NoSuchGuestException` exception.

    In one sentence, name one major problem with this design and why it is a problem.

> **Solution:** There are several issues. You just need to name one.
>
> - The programmer is using exception as flow control. A better way would be for `lookUpGuest` to return a boolean to indicate if a guest exists.
> - The second `catch` catches all possible exceptions. This is the Pokémon exception and should be avoided.
> - `NoSuchGuestException` is something that the programmer anticipates so it should be a checked exception instead.

Generics (9 marks)

Ah Beng wrote the following class:

```
import cs2030s.fp.Transformer;

class Sure<T> {
  private T x;

  private Sure(T x) {
    this.x = x;
  }

  public static <T> Sure<T> of(T x) {
        return new Sure<T>(x);
  }

  // map inserted here
}
```

He wanted to add a method `map` to the class above. For each possible method header of `map` below, indicate which of the corresponding statement(s) would lead to a compilation error. You can assume that Ah Beng implemented the method `map` correctly so that there is no compilation error in the class `Sure<T>`.

11. If Ah Beng declared the method `map` as follows:

```
public <R> Sure<?> map(Transformer<T, ? extends R> f) {
  :
}
```

Which of the following statement(s) would lead to a compilation error?

```
Sure<?> s1 = Sure.of("hello").map(str -> str.length());         // Statement A

Sure<Integer> s2 = Sure.of("hello").map(str -> str.length());  // Statement B

Transformer<String, Number> str2len = str -> str.length();
Sure<Integer> s3 = Sure.of("hello").map(str2len);               // Statement C
```

Solution: The lambda expression `str -> str.length()` is passed as parameter to `map`. Through type inference, `T` will have the type of `String`, and `R` will have the type `Integer`. Statement A compiles without error.

`Sure<?>` is a supertype of `Integer`. Statements B and C cannot compile.

12. If Ah Beng declared the method `map` as follows:

```
public Sure<? extends Number> map(Transformer<T, ? extends Object> f) {
  :
}
```

Which of the following statement(s) would lead to a compilation error?

```
Sure<?> s1 = Sure.of("hello").map(str -> str.length());          // Statement A

Sure<Integer> s2 = Sure.of("hello").map(str -> str.length());    // Statement B

Transformer<String, Number> str2len = str -> str.length();
Sure<Integer> s3 = Sure.of("hello").map(str2len);                // Statement C
```

Solution: Consider Expression A. The lambda expression `str -> str.length()` is passed as parameter to `map`. Through type inference, `T` will have the type of `String`. `str.length()` returns an `int` and so the expression `str -> str.length()` matches the type:

`Transformer<T, ? extends Object>`.

The return type is `Sure<? extends Number>`, which is a subtype of `Sure<?>`. So, Statement A compiles without error.

`Sure<? extends Number>` is a supertype of `Integer`. Statements B and C cannot compile.

13. If Ah Beng declared the method `map` as follows:

```
public <R> Sure<R> map(Transformer<T, R> f) {
  :
}
```

Which of the following statement(s) would lead to a compilation error?

```
Sure<?> s1 = Sure.of("hello").map(str -> str.length());          // Statement A

Sure<Integer> s2 = Sure.of("hello").map(str -> str.length());    // Statement B

Transformer<String, Number> str2len = str -> str.length();
Sure<Integer> s3 = Sure.of("hello").map(str2len);                // Statement C
```

Solution: Consider Expression A. The lambda expression `str -> str.length()` is passed as parameter to `map`. Through type inference, `T` will have the type of `String` and `R` the type `Integer`. The return type is `Sure<R>`, which is `Sure<Integer>`, which is a subtype of `Sure<?>`. So, Statement A, again, compiles without error. So is Statement B.

For Statement C, if `R` is inferred to be `Integer`, the return type matches, but the type of the parameter is incorrect (`Transformer<String, Number>` instead of `Transformer<String, Integer`, which are invariant). But if `R` is inferred to be `Number`, the return type does not match (`Sure<Number>` instead of `Sure<Integer>`, which are again invariant). So Statement C cannot compile.

Streams (8 marks)

This question will require you to create a stream pipeline by selecting each component of the pipeline. You are to create a stream that contains the following elements:

1A, 1B, 2A, 2B, 4A, 4B, 5A, 5B, 7A, 7B, .., 14A, 14B

Note: any number that is divisible by 3 is not present in the stream.

You may use the following list in your pipeline:

```
List<String> strings = List.of("A", "B");
```

Some helpful stream methods:

- `Stream<T> filter(Predicate<? super T> predicate)` Returns a stream consisting of the elements of this stream that match the given `predicate`.
- `<R> Stream<R> map(Function<? super T, ? extends R> mapper)` Returns a stream consisting of the results of applying the given function to the elements of this stream.
- `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)` Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
- `Stream<T> takeWhile(Predicate<? super T> predicate)` Returns, if this stream is ordered, a stream consisting of the longest prefix of elements taken from this stream that match the given `predicate`. Otherwise returns, if this stream is unordered, a stream consisting of a subset of elements taken from this stream that match the given predicate.
- `Stream<T> limit(long maxSize)` Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.
- `static <T> Stream<T> of(T... values)` Returns a sequential ordered stream whose elements are the specified values.
- `static <T> Stream<T> generate(Supplier<? extends T> s)` Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
- `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)` Returns an infinite sequential ordered `Stream` produced by iterative application of a function `f` to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc. (Note `UnaryOperator<T>` is equivalent to `Function<T, T>`)

You can convert a `List` instance to a `Stream` using the method `List::stream()`, and from a `Stream` to a `List` using the method `Stream::collect` (passing in `Collectors.toList()` as argument).

14. First, choose the data source:

    A. `Stream.generate(() -> x + 1)`

    B. `Stream.of(1, 2, 4, 5, ..)`

    C. `Stream.iterate(1, x -> x + 1)`

    D. `Stream.of("A", "B")`

15. Choose the next intermediate operation:

    A. `.filter(i -> i % 3 == 0)`

B. `.filter(i -> i % 3 != 0)`

C. `.flatMap(y -> strings.stream().map(z -> y + z))`

D. `.map(y -> strings.stream().flatMap(z -> y + z))`

16. Choose the next intermediate operation:

A. `.flatMap(y -> strings.stream().map(z -> y + z))`

B. `.map(y -> strings.stream().flatMap(z -> y + z))`

C. `.filter(i -> i % 3 == 0)`

D. `.filter(i -> i % 3 != 0)`

17. Choose the terminal operation:

A. `.collect(Collectors.toList());`

B. `.takeWhile(r -> r < 14);`

C. `.limit(20);`

D. `.limit(14);`

Solution: The full pipeline is:

```
Stream.iterate(1, x -> x + 1)
  .filter(i -> i % 3 != 0)
  .flatMap(y -> strings.stream().map(z -> y + z))
  .limit(20)
```

So the answer is C, B, A, C.

Erasure (4 marks)

18. Consider the class below.

```
public class Dictionary<T extends Comparable<T>, S> {
    Maybe<T> key;
    S value;

    public Dictionary(Maybe<T> key, S value) {
        this.key = key;
        this.value = value;
    }
}
```

What is the type of `key` after type erasure?

A. `Object`

B. `Comparable<Maybe>`

C. `Maybe`

D. `Comparable`

E. `Maybe<Comparable>`

F. `Maybe<Object>`

Solution: `key` is a parameterized type. Type erasure removes all type arguments and type parameters of parameterized types, so the type of `key` after erasure is `Maybe`.

19. Consider the class below.

```
public class Dictionary<T extends Comparable<T>, S> {
    Maybe<T> key;
    S value;

    public Dictionary(Maybe<T> key, S value) {
        this.key = key;
        this.value = value;
    }
}
```

What is the type of `value` after type erasure?

A. `Object`

B. `S`

C. `Array`

D. `Value`

E. `T`

F. `Comparable`

Solution: `value` has the type `S`, which is an unbounded type parameter. All unbounded type parameter are replaced with `Object` after erasure.

Lambdas (8 marks)

Consider the following simple monad `Box<T>`. As we have learned, a `Box<T>` can store a value of any type inside.

```
import cs2030s.fp.Transformer;

class Box<T> {
  private T x;

  private Box(T x) {
    this.x = x;
  }

  public static <T> Box<T> of(T x) {
    return new Box<>(x);
  }

  public T get() {
    return x;
  }

  public <R> Box<R> map(Transformer<? super T, ? extends R> t) {
    return new Box<>(t.transform(x));
  }
}
```

Since functions, in the form of lambda expressions, are first-class citizens in Java, we can store a lambda expression in a `Box<T>` too.

20. Consider the following two transformers:

```
Transformer<Integer, Integer> f = x -> x + 1;
Transformer<Integer, Integer> g = x -> x * 2;
```

We put `f` in a `Box<T>` object:

```
Box<Transformer<Integer, Integer>> box;
box = Box.of(f);
```

We can now use `map` to transform our lambda expressions. First, let's suppose we want to compose the function inside the box with another function.

Show how you can compose `g` with `f` using `map` to get the function `x -> (x + 1) * 2` by filling in the missing argument of the `map` method below.

```
box = box.map(???);
box.get().transform(4); // should return 10;
```

Note that you should not hard-code the function `x -> (x + 1) * 2` in your answer. You answer should work for any `g` and for any function in `box`.

Solution: `ff -> x -> g.transform(ff.transform(x))`

21. In our module, we have two functional interfaces to represent functions, `cs2030s.fp.Transformer`, and `java.util.function.Function`. It is sometimes useful to change between the two.

Fill in the missing argument `???` to `map` in the line of code below, which converts the lambda expression of type `Transformer<Integer, Integer>`, to `Function<Integer, Integer>`.

```
Transformer<Integer, Integer> incr = x -> x + 1;
Box<Transformer<Integer, Integer>> box;
box = Box.of(incr);
Box<Function<Integer, Integer>> box2 = box.map(???);
```

The functional method for `java.util.function.Function` is called `apply` (you may or may not need this information to answer this question).

Solution: `ff -> x -> ff.transform(x)`

Monad (10 marks)

Consider the following monad, `Monad<T>`,

```
import cs2030s.fp.Transformer;

class Monad<T> {
  private T x;

  private Monad(T x) {
    this.x = x;
  }

  public static <T> Monad<T> of(T x) {
    return new Monad<>(x);
  }

  public T get() {
    return x;
  }

  public <R> Monad<R> flatMap(Transformer<? super T, ? extends Monad<? extends R>> f) {
    return new Monad<>(f.transform(this.x).get());
  }

  public <R> Monad<R> map(Transformer<? super T, ? extends R> f) {
        return flatMap(???);
  }
}
```

22. Complete the implementation of `map` using only `flatMap` so that the resulting `Monad<T>` that satisfies the functor laws.

Solution: `x -> Monad.of(f.transform(x))`

23. Show that the monad `Monad<T>` preserves composition and therefore meets one of the requirements of being a functor. The skeleton of the proof is given below. Fill in the blanks by completing the Expressions A, B, and C, as well as two monad laws we use.

Suppose we have an instance of `Monad<T>` called `m` and two functions `f` and `g`.

`m.map(x -> f(x)).map(x -> g(x))`

is equivalent to the following Expression A based on the implementation above:

`m.flatMap(x -> Monad.of(f(x))).flatMap(x -> Monad.of(g(x)))` (Expression A)

Invoking Monad's **Associative** Law, Expression A is equivalent to Expression B below,

`m.flatMap(x -> Monad.of(f(x)).flatMap(x -> Monad.of(g(x))))` (Expression B)

Invoking Monad's **Left Identity** Law, Expression B is equivalent to Expression C below,

`m.flatMap(x -> Monad.of(g(f(x))))` (Expression C)

which, by our implementation, is equivalent to

`m.map(x -> g(f(x)))`

Therefore, the composition of functions is preserved in our implementation.

CompletableFuture (8 marks)

24. We have an incomplete program below:

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

class Main {

  private static void doSomething() {
     :   runs for some time
  }

  private static Function<Integer,Integer> plus(int i) {
    return x -> {
      doSomething();
      System.out.println(x + i);
      return x + i;
    };
  }

  public static void main(String args[]) {
    CompletableFuture<Integer> ten =
        CompletableFuture.supplyAsync(() -> 10);

    // Line X

  }
}
```

The method `doSomething()` runs for an indeterministic amount of time. Its method body has been omitted.

Consider the following code snippets to be inserted into the method `main` after the declaration of variable `ten` (starting at the line marked `Line X`) and the possibilities of what could be printed. You may assume that `doSomething()` does not print anything.

Select all options that are TRUE.

A. If we write

```
ten.thenApply(plus(1)).thenApply(plus(10)).thenApply(plus(5));
```

the program will always print

```
11
21
26
```

every time it is executed.

> **Solution:** The intended answer is FALSE. The `CompletableFuture` is not join()-ed. So the program can exit without all three numbers printed. If anything is printed, it will be in the order shown.
>
> However, in the toy program we wrote, there are no other tasks. It appears that Java VM will schedule the `main` thread to run `() -> 10` (and therefore, the rest of the `plus` calls). So, despite having not having a `join()`, the program will still exit after the `main` thread exits, and all three numbers will be printed in order.

For this option, we will accept both TRUE and FALSE as correct.

B. If we write

```
CompletableFuture<Integer> cf = ten.thenApply(plus(1));
cf.thenApply(plus(10)).join();
cf.thenApply(plus(5)).join();
```

the program will always print

```
11
21
16
```

every time it is executed.

Solution: TRUE. `join()` ensures that the program exists only after all three numbers are printed. Due to the `join()` in the second line, the third line will always be executed after the second line completes. Thus, the numbers will always be printed and printed in order.

C. If we write

```
CompletableFuture<Integer> cf = ten.thenApply(plus(1));
cf = cf.thenApplyAsync(plus(10));
cf.thenApplyAsync(plus(5)).join();
```

the program will always print

```
11
21
26
```

every time it is executed.

Solution: TRUE. `join()` ensures that the program exists only after all three numbers are printed. Since the completable future chains them in order, the numbers will be printed in order.

D. If we write

```
CompletableFuture.allOf(
   ten.thenApplyAsync(plus(1)),
   ten.thenApplyAsync(plus(10)),
   ten.thenApplyAsync(plus(5))
).join();
```

the program will always print

```
11
20
15
```

every time it is executed.

Solution: FALSE. `join()` only ensures that the program exists only after all three numbers are printed, but not the order of the numbers are printed.

Fork and Join (3 marks)

Consider the following variation of the class `Summer`, where we split the task into three smaller subtasks: `left`, `mid`, and `right`.

```
import java.util.concurrent.RecursiveTask;

class Summer extends RecursiveTask<Integer> {
  private static final int FORK_THRESHOLD = 5;
  private int low;
  private int high;
  private int[] array;

  public Summer(int low, int high, int[] array) {
    this.low = low;
    this.high = high;
    this.array = array;
  }

  @Override
  protected Integer compute() {
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int onethird = low + (high - low) / 3;
    int twothird = onethird + (high - low) / 3;

    Summer left = new Summer(low, onethird, array);
    Summer mid = new Summer(onethird, twothird, array);
    Summer right = new Summer(twothird, high, array);

      :
  }
}
```

25. Which of the following options achieves the highest level of parallelism?

A.
```
left.fork();
right.fork();
return right.join() + left.join() + mid.compute();
```

B.
```
right.fork();
left.fork();
return mid.compute() + left.join() + right.join();
```

C.
```
mid.fork();
return left.compute() + mid.join() + right.compute();
```

D.
```
right.fork();
left.fork();
return left.join() + mid.compute() + right.join();
```

Solution: First, note that the highest level of parallelism does not mean the most efficient, due to the overhead of creating a task.

Option A runs `mid` sequentially after `left` and `right`.

Option B runs `left`, `right`, and `mid` parallelly.

Option C runs `right` sequentially after `mid` and `left`.

Finally, option D runs `mid` sequentially after `left`.

So the B has the highest level of parallelism.

END OF PAPER