NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
MIDTERM ASSESSMENT FOR
Semester 2 AY2021/2022

CS2030S Programming Methodology II

February 2022 Time Allowed 70 minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 27 questions and comprises 15 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 a **OPEN BOOK** assessment. You are only allowed to refer to hardcopies materials.
5. All questions in this assessment paper use Java 11.

Section I: Java Basics (6 points)

Select the most appropriate answer. 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.

1. (1 point) Consider the following code excerpt:

   ```
   Building house = new House();
   ```

   What is the Compile-Time type of `house`?

   A. `Building`

   B. `House`

   C. `Object`

   D. `null`

   E. None of the above.

   Solution: A

2. (1 point) Consider the same code excerpt as the previous question:

   ```
   Building house = new House();
   ```

   What is the Run-Time type of `house` after the excerpt above is executed?

   A. `Building`

   B. `House`

   C. `Object`

   D. `null`

   E. None of the above.

   Solution: B

3. (1 point) Which of the following statements is not true about Java?

   A. The Java compiler `javac` compiles Java source to bytecode.

   B. Java does not allow multiple inheritance for classes.

   C. The Java Virtual Machine executes Java bytecode.

   D. Java is a dynamically typed language.

   E. All statements about Java above are correct.

   Solution: D or E

4. (1 point) Consider the following code excerpt:

```java
public class A {
    public int foo(int i) {
        return i;
    }
}

public class B extends A {
    public int foo(int i, int j) {
        return i + j;
    }
}
```

The Java feature that allows both `foo` in class `A` and foo in class `B` to be defined is

A. method overloading.
B. method overriding.
C. method overwriting.
D. method chaining.
E. None of the above.

Solution: A

5. (1 point) Consider the following code excerpt:

```java
Box<String> life = Box.<String>of("Chocolates");
```

The line above compiles without errors. The method `of` returns a new instance of `Box<T>`.

Select all terms which correctly describe the method `of`.

i. A generic method
ii. A class method
iii. A factory method
iv. A constructor

Solution: (i), (ii) and (iii) only

6. (1 point) Consider the following code excerpt:

```
Point p1 = new Point(1, 1);
Point p2 = new Point(2, 2);
Circle circle = new Circle(p1, 3);
circle.contains(p2);
```

What is the *target* of invocation of the method `contains`?

A. `p1`

B. `p2`

C. `circle`

D. `null`

E. None of the above.

Solution: C

Section II: Polymorphism (8 points)

Consider the following classes:

```
class A {
    public void doTask(A a) { }
}

class B extends A {
    @Override
    public void doTask(A a) { }

    public void doTask(B b) { }
}

class C extends B {
    public void doTask(C c) { }
}

class D extends B {
    @Override
    public void doTask(A a) { }
}
```

We use the method signature notation `doTask(X)`, where `X` is the compile-time type of the parameter, to differentiate the multiple `doTask` methods defined in a class.

Consider the code excerpt:

```
A a = new A();
B b = new B();
A c = new C();
D d = new D();
```

7. (2 points) Consider the following code excerpt:

```
a.doTask(d);
```

Which `doTask` will be invoked?

A. `doTask(A)` in class `A`

B. `doTask(A)` in class `B`

C. `doTask(B)` in class `B`

D. `doTask(C)` in class `C`

E. `doTask(A)` in class `D`

F. None of the above.

Solution: `doTask(A)` in class `A`. The compile-time type of `a` is `A`. The selected method would be `void doTask(A)`. During run-time, `a` has the run-time type of `A`. Since there is a method with a matching descriptor in `A`, the method in `A` is invoked.

8. (2 points) Consider the following code excerpt:

```
b.doTask(c);
```

Which `doTask` will be invoked?

A. `doTask(A)` in class `A`

B. `doTask(A)` in class `B`

C. `doTask(B)` in class `B`

D. `doTask(C)` in class `C`

E. `doTask(A)` in class `D`

F. None of the above.

Solution: `doTask(A)` in class B. The compile-time type of `b` and `c` are `B` and `A` respectively. Even though there are two `doTask` accesssible in `B`, The only method that we can apply to `c` as argument is `void doTask(A)`. During run-time, `b` has the run-time type of `B`. Since there is a method with a matching descriptor in `B`, the method `doTask(A)` in `B` is invoked.

9. (2 points) Consider the following code excerpt:

```
c.doTask(d);
```

Which `doTask` will be invoked?

A. `doTask(A)` in class `A`

B. `doTask(A)` in class `B`

C. `doTask(B)` in class `B`

D. `doTask(C)` in class `C`

E. `doTask(A)` in class `D`

F. None of the above.

Solution: `doTask(A)` in class `B`. The compile-time type of `c` and `d` are `A` and `D` respectively. There is only one `doTask` method in `A`, so the method descriptor `void doTask(A)` is chosen. During run-time, `c` has the run-time type of `C`. But there is no method `doTaskA` defined in `C`, travelling up the class hierarchy, `doTask(A)` is found in class `B`, so it is invoked.

10. (2 points) Consider the following code excerpt:

```
d.doTask(c);
```

Which `doTask` will be invoked?

A. `doTask(A)` in class `A`
B. `doTask(A)` in class `B`
C. `doTask(B)` in class `B`
D. `doTask(C)` in class `C`
E. `doTask(A)` in class `D`
F. None of the above.

Solution: `doTask(A)` in class `D`. The compile-time type of `d` and `c` are `D` and `A` respectively. There is only three accessible `doTask` methods in `D`, but the only method descriptor that applies to `A` as argument is `void doTask(A)`. Again, this method descriptor is chosen. During run-time, `d` has the run-time type of `D`. Since there is a method with matching descriptor in `D`, it is invoked.

Section III: Exceptions (4 points)

Consider the following Exception hierarchy and `try-catch` block:

`CException` <: `BException` <: `AException`

```
try {
    method();
} catch (CException e) {
    // Block A
} catch (AException e) {
    // Block B
}
```

11. (2 points) Which of the above blocks would be executed if `method()` threw a `CException`?

A. Block A and then Block B.
B. Block A only.
C. Block B only.
D. None of the above.

Solution: Block A only

12. (2 points) Which of the above blocks would be executed if `method()` threw a `BException`?

   A. Block A and then Block B.

   B. Block A only.

   C. Block B only.

   D. None of the above.

Solution: Block B only

Section IV: Type Inference (6 points)

Consider the following classes:

```
class Location {}
class Country extends Location {}
class CityState extends Country {}
```

13. (3 points) Now, consider the following method:

```
<T> T largestArea(Array<? extends T> array) {
  // method body omitted
}
```

Then, consider following code excerpt:

```
Location l = largestArea(new Array<CityState>(0));
```

What is `T` inferred as?

   A. `Object`

   B. `?`

   C. `Location`

   D. `Country`

   E. `CityState`

   F. No possible type satisfies `T`, it will not compile.

Solution: The return type constrains `S` <: `Location`, and the argument `array` constrains `S` >: `CityState`. The most specific type that meets these constraints is `CityState`.

14. (3 points) Now, consider the following method:

```
<S> S getFurthest(Array<? extends S> p1, Array<? super S> p2) {
  // method body omitted
}
```

Then, consider following code excerpt:

```
Location p = getFurthest(new Array<Location>(0), new Array<CityState>(0));
```

What is `S` inferred as?

A. `Object`

B. `?`

C. `Location`

D. `Country`

E. `CityState`

F. No possible type satisfies `S`, it will not compile.

Solution: The return type constrains `S` <: `Location`, the argument `p1` constrains `S` >: `Location`, and the argument `p2` constrains `S` <: `CityState`. No possible type satisfies `S`, it will not compile

Section V: Type System (9 points)

15. (3 points) Consider the following subtyping relationships:

`A2` <: `A1`
`A4` <: `A1`
`A4` <: `A2`
`A3` <: `A2`

Select all of the following class and interface structures below which match all four of the subtyping relationships above. Note that the correct options may satisfy additional subtyping relationships.

A.
```
interface A1 {}
class A2 implements A1 {}
class A3 extends A2 {}
class A4 extends A3 {}
```

B.
```
interface A1 {}
class A2 implements A1 {}
class A3 extends A2 {}
class A4 implements A1 {}
```

C.
```
interface A1 {}
interface A2 extends A1 {}
interface A3 extends A2 {}
class A4 implements A3 {}
```

D.
```
class A1 {}
class A2 extends A1 {}
class A3 extends A2 {}
class A4 extends A1 {}
```

Solution: A, C

16. (6 points) Consider the following subtyping between classes `B1`, `B2`, `B3`, and Interface `I1`:

    `B3` <: `B2` <: `B1`
    `B2` <: `I1`

    Consider the following method in the `Box` class:

```java
class Box<T> {
  private T x;
  private Box(T x) {
    this.x = x;
  }
  public static <T> Box<T> of(T t) {
    if (t == null) {
      return null;
    }
    return new Box<>(t);
  }
}
```

and the following code excerpt:

```java
box = Box.of(new B2());
```

What are the possible valid compile-time types for `box` so that the statement above compiles without any warning or error?
Give eight possibilties for full marks.

Solution: An object of type `B2` is passed into `Box`, so `box` could have the type of `Box<B2>` (i.e., `T` is inferred as `B2`).

We can always assign a value of type `Box<B2>` to a variable that is a supertype of `Box<B2>`. Supertypes of `Box<B2>` includes:

- `Box<?>`
- `Box<? extends B2>`
- `Box<? super B2>`
- `Box<? super B3>` (by contravariance)
- `Box<? extends B1>` (by covariance)
- `Box<? extends I1>` (by covariance)
- `Box<? extends Object>` (by covariance)

`box` could have the type `Box<B1>` as well. In this case, `T` is inferred as `B1`. Similarly, we can assign a value of type `Box<B1>` to a variable that is a supertype of `Box<B1>`. This adds the type `Box<? super B1>` to the list.
Similarly, `box` could have the type `Box<I1>` (`T` is inferred as `I1`) and we can add the type `Box<? super I1>` to the list.
Ditto for `Box<Object>` and `Box<? super Object>`.

Finally, `box` could be `Object`.

So, there are plenty of choices.

Among all possible combinations of bounded wildcards and the relevant types above, only `Box<B3>` and `Box<? extends B3>` are wrong.

Section VI: OOP Principles (4 points)

Consider the following Java program:

```java
public class BankAccount {
    double balance;

    private void setBalance(double balance) {
        this.balance = balance;
    }
}

public class Customer {
    BankAccount account = new BankAccount();

    public boolean withdraw(double amount) {
        if (account.balance > amount) {
            account.balance -= amount;
            return true;
        }
        return false;
    }
}
```

17. (2 points) Does this program follow the principle of information hiding? Explain your reasoning in no more than two sentences.

Solution: This program does not follow the principle of information hiding. The balance information in `BankAccount` is publically accessible.

18. (2 points) Does this program follow the principle of "Tell, Don't Ask?" Explain your reasoning in no more than two sentences.

Solution: This program does not follow the principle of Tell, Don't Ask, as the `Customer` class directly checks the balance of the `BankAccount` and then modifies the value.

Section VII: Generics (9 points)

19. (3 points) The following code would compile without any syntax error or warning. True or false? Explain.

```
class A<T> {
  public T foo() {
    return null;
  }

  public Object foo() {
    return null;
  }
}
```

Solution: False.

Two possible answers:

- Both `foo` have the same method signatures and this is not allowed.
- `T` is erased to `Object`. So both `foo` methods has the same method descriptor after erasure and is not allowed.

20. (3 points) The following code would compile without any syntax error or warning. True or false? Explain.

```
class A<T extends Comparable<T>> {
  public void foo(T t) {
  }

  public void foo(Object o) {
  }
}
```

Solution: True. `T` is erased to `Comparable`. So the two `foo` methods does not have the same method signature and are considered as method overloading.

21. (3 points) The following code would compile without any syntax error or warning. True or false? Explain.

```
class A<T extends Comparable<T>> {
  public T foo() {
    return null;
  }
}

class B<T> extends A<T> {
  public T foo() {
    return null;
  }
}
```

Solution: False. Two possible answers:

- The `T` in `A` requires `T` to be a subclass of `Comparable<T>`. But the `T` in `B` does not have such as constraint so the `T` from `B` cannot be passed as type argument to `A`.
- Alternatively, `T` in `A` is erased to `Comparable` but `T` in `B` is erased to `Object`. The return type of the overriding `foo` cannot be a supertype of the return type of the overridden `foo`.

Section VIII: Method Overriding (7 points)

In this question, we extend the definition of subtyping to a tuple of types. Given two tuples of types $(S_1, S_2, ..S_m)$ and $(T_1, T_2, ..T_n)$, we say that

$$(S_1, S_2, ..S_m) <: (T_1, T_2, ..T_n)$$

if and only if $m = n$ and $S_i <: T_i$ for all $1 \leq i \leq m$.

We can treat the types of the parameters to a method as a tuple of types. For example, the types of the parameters of method

```
void foo(Object o, Number n) { }
```

can be treated as the tuple (`Object` , `Number`). With the new definition, we can say whether the parameters of one method is a subtype of the parameters of the other methods.

A community of programmers have come together to build a new object-oriented programming language called NeoJava, which follows the syntax and behaviour of Java but, being a new language, NeoJava can forgo backward compatibility constraints that has bugged Java. NeoJava now accepts proposals to change the behavior of the language.

Now, consider the rules of method overriding in the current version of Java. A method m_1 defined in class `A` is overridden by a method m_2 in class `B` (which is a subclass of `A`) if m_2 and m_1 has the same method signature and the return type of m_2 is a subtype of the return type of m_1.

22. (5 points) Ah Kow would like to make overriding in NeoJava more flexible than Java, and wrote the following proposal to the NeoJava Design Committee.

”I would like to propose that we relax the rule on method overriding, and allow a method m_1 defined in class `A` to be overriden by a method m_2 in class `B` (which is a subclass of `A`) as long as (i) m_2 and m_1 has the same name; (ii) the return type of m_2 is a subtype of the return type of m_1, and (iii) the parameters of m_2 is a subtype of the parameters of m_1.”

Ah Kow gave the following example, which would compile under his proposal:

```
class A {
  void f(A a1, A a2) {
  }
}

class B extends A {
  @Override
  void f(B b1, B b2) {
  }
}
```

The committee rejected Ah Kow’s proposal outright.

Which principle does Ah Kow’s proposal violate? Explain your answer with an example of how the classes written by Ah Kow’s above could be used.

Solution: It violates LSP. Consider the code

```
void g(A a) {
  a.foo(new A(), new A());
}
```

which works with `A` . But when an instance of `B` is passed into `g` , it would not work, since we are now passing A as B into `foo` .

23. (2 points) Undeterred, Ah Kow submit a new proposal, changing from ”subtype” to ”supertype” in condition (iii):

    ”I would like to propose that we relax the rule on method overriding, and allow a method m_1 defined in class `A` to be overriden by a method m_2 in class `B` (which is a subclass of `A`) as long as (i) m_2 and m_1 has the same name; (ii) the return type of m_2 is a subtype of the return type of m_1, and (iii) the parameters of m_2 is a *supertype* of the parameters of m_1.”

    Ah Kow gave the following new example, which would be compile under his new proposal:

```
class A {
  void f(A o1, B o2) {
  }
}

class B extends A {
  @Override
  void f(Object a1, Object a2) {
  }
}
```

    Does it still violate the same principle as his previous proposal violates? Explain in no more than two sentences.

Solution: It does not violate LSP anymore. Consider the code

```
    void g(A a) {
      a.foo(new A(), new A());
    }
```

It would still work if an instance of `B` is passed into `g`. `B::foo` only expects `Object` instances and `A` s are valid instances of `Object` s.

Section IX: Type Checking (9 points)

Consider the following code excerpt:

```java
class Store<T> {
  T x;
  void keep(T x) {
    this.x = x;
  }
  T get() {
    return this.x;
  }
}

Store<String> stringStore = new Store<>();
Store store = stringStore;

store.keep(123);  // Line A

String s = stringStore.get(); // Line B
```

24. (3 points) Why does the Java compiler give a compilation warning in Line A?

Solution: The compiler will give us an unchecked warning, as the compiler is not sure if this line is a type safe operation. Specifically, as we are using the Raw Type `Store`, the compiler can not check if it is safe to pass an `Integer` to the `keep` method. It will allow us to do this, but warn the programmer

25. (3 points) What will happen during run time in Line A? Explain your answer in no more than two sentences.

Solution: During runtime, the program will assign the value 123 to the variable x. This is allowed due to type erasure, i.e. `T x` will become `Object x` after type erasure, and `Object` can point to an `Integer`.

26. (3 points) What will happen during run time in Line B? Explain your answer in no more than two sentences.

Solution: The program will have a ClassCastException as it tries to cast an `Integer` to a `String`. This is the result of the previous line in which we did an unsafe operation.

Section X: Stack and Heap (8 points)

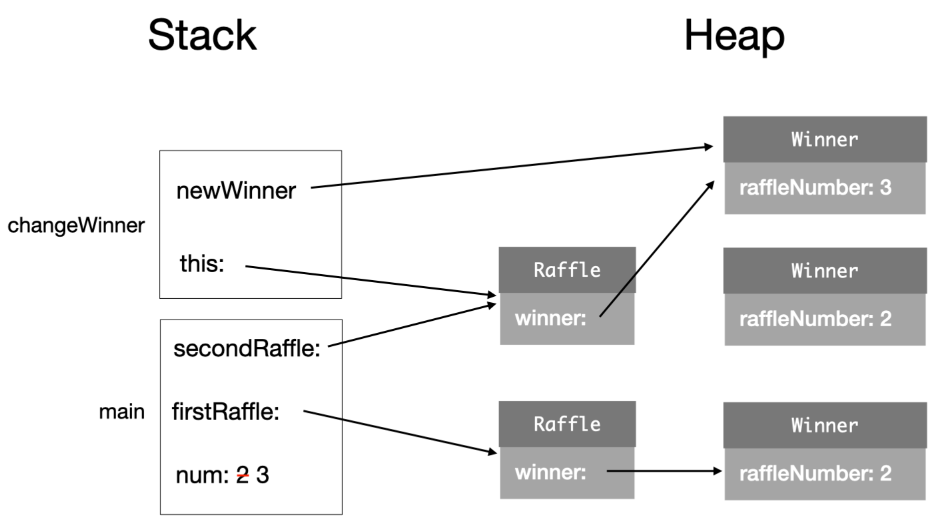
27. (8 points) Consider the following Java classes:

```
class Winner {
    private int raffleNumber;

    public Winner(int raffleNumber) {
        this.raffleNumber = raffleNumber;
    }
}

class Raffle {
    private Winner winner;

    public Raffle(Winner winner) {
        this.winner = winner;
    }

    public void changeWinner(Winner newWinner) {
        this.winner = newWinner;
        // Line A
    }
}
```

Now consider the following code excerpt from a program's `main` method.

```
int num = 2;
Raffle firstRaffle = new Raffle(new Winner(num));
Raffle secondRaffle = new Raffle(new Winner(2));
num += 1;
secondRaffle.changeWinner(new Winner(num));
```

The program `main` method will be executed. On the provided sheet, draw the Stack and the Heap after the program executes the `main` method up to Line A. Do not remove any objects created on the heap. You may ignore any variables used in `main` but not shown in the excerpt above.



Solution:

END OF PAPER