NATIONAL UNIVERSITY OF SINGAPORE

CS2030S– Programming Methodology II

MIDTERM (ANSWER)

September 2022 TIME ALLOWED: 70 MINS

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains **TWENTY SEVEN (27)** questions and comprises **TWENTY (20)** printed pages.
2. Write all your answers in the answer sheet provided.
3. The total mark for this assessment is 70.
4. This is an **OPEN BOOK** assessment. You are only allowed to refer to hardcopies materials.
5. All questions in this assessment paper uses Java 17.
6. Answer **ALL** questions.
7. Question types:
    - **MCQ:** Select the most appropriate answer. If multiple answers are equally appropriate, pick one and write the chosen answer (*i.e., the letters*) in the answer box. Do NOT write more than one answer for MCQ.
    - **MRQ:** Select ALL correct answers. Write ALL the chosen answers (*i.e., the letters*) in the answer box. No partial mark given.
    - **ESSAY:** Write your answer in the box provided.
    - **TRUE/FALSE:** Write True or False as well as any explanations needed.

Section I: Java Basic (6 points)

1. **[MRQ]** *(1 point)* Select ALL statements that are true about Java type systems.

   A. ***Java is statically typed***

   B. Java is dynamically typed

   C. ***Java is strongly typed***

   D. Java is weakly typed

Java is strongly and statically typed language.

2. **[MRQ]** *(1 point)* Select ALL constructs that is resolved via static binding instead of dynamic binding.

   A. Non-static method

   B. ***Static method***

   C. ***Non-static field***

   D. ***Static field***

Only non-static method is resolved via dynamic binding

3. **[MRQ]** *(1 point)* Select ALL statements that are false about Java `final` keyword.

   A. ***`final` keyword can be used to prevent a single method from being inherited***

   B. ***`final` keyword can be used to prevent a single method from being overloaded***

   C. `final` keyword can be used to prevent a single method from being overridden

   D. `final` keyword can be used to prevent a single class from being inherited

(A) is wrong because it can prevent the class from being inherited, not a single method. (B) is wrong because it prevents overriding and not overloading.

4. [**MRQ**] *(1 point)* Consider the following classes

```
class A {
  public void foo(A param) { .. }
}
class B {
  public void foo(A param) { .. }
  public void foo(B param) { .. }
}
```

Select ALL concepts that are used in the code above.

A. Method overriding

B. ***Method overloading***

C. Inheritance

D. Polymorphism

No inheritance because we do not use the `extends` keyword, so no inheritance and polymorphism also. `foo(A)` and `foo(B)` is overloading.

5. [**MRQ**] *(1 point)* Select ALL statements that are true about complex type in Java.

A. Java generic is covariant

B. Java generic is contravariant

C. ***Java generic is invariant***

D. ***Java array is covariant***

E. Java array is contravariant

F. Java array is invariant

6. [**MRQ**] *(1 point)* Select ALL the statements that are true regarding access modifier in Java.

A. ***Fields/methods with `private` access modifier can be accessed by code inside the class***

B. Fields/methods with `private` access modifier can be accessed by code in the subclass

C. ***Fields/methods with `public` access modifier can be accessed by code inside the class***

D. ***Fields/methods with `public` access modifier can be accessed by code in the subclass***

Section II: Interface and Abstract Class (4 points)

For this section, you are given the following interface

```
interface I1 {
  void f1();
  void g();
}
interface I2 {
  void f2();
  void g();
}
interface I3 extends I1, I2 { }
```

Additionally, you are given the following classes where the `abstract` modifier is omitted.

```
/* abstract modifier omitted */ class C1 implements I1 {
  public void g() {  }
}
/* abstract modifier omitted */ class C2 implements I2 {
  public void f1() {  }
  public void g() {  }
}
/* abstract modifier omitted */ class C3 implements I3 {
  public void f2() {  }
  public void g() {  }
}
/* abstract modifier omitted */ class C4 extends C2 implements I1 {
  public void f2() {  }
}
```

7. [**MRQ**] *(2 points)* Select ALL classes that *must* have the `abstract` modifier in the class declaration.

   A. ***Class C1***

   B. ***Class C2***

   C. ***Class C3***

   D. Class `C4`

   A class *must* be `abstract` when it has an abstract method.

   - Class `C1` does not have the implementation for `void f1()` in `I1` so it is still abstract.
   - Class `C2` does not have the implementation for `void f2()` in `I2` so it is still abstract.
   - Class `C3` does not have the implementation for `void f1()` in `I3` (*which itself inherits from* ***I1***) so it is still abstract.
   - Class `C4` has an implementation for `void f1()` and `void g()` inherited from `C2` and an implementation for `void f2()` in its own class. So this is the only class that need not be abstract.

8. [**MRQ**] *(2 points)* Assume that all abstract modifiers have been added correctly such that the code compiles. Select ALL valid subtype relationships.

   A. `C1 <: I2` B. `C2 <: I1` C. ***C3 <: I1*** D. ***C4 <: I2***

   The subtyping relationships are:

   - `C1 <: I1` (*from* ***C1 implements I1***)
   - `C2 <: I2` (*from* ***C2 implements I2***)
   - `C3 <: I3` (*from* ***C3 implements I3***)
   - `I3 <: I1` and `C3 <: I1` (*from* ***I3 extends I1*** *and transitivity*)
   - `I3 <: I2` and `C3 <: I2` (*from* ***I3 extends I2*** *and transitivity*)
   - `C4 <: C2 <: I2` (*from* ***C4 extends C2***)
   - `C4 <: I1` (*from* ***C4 implements I1***)

   So the correct answers are (C) and (D).

Section III: Inheritance, Polymorphism, and Dynamic Binding (8 points)

For this section, you are given the following classes where the `@Override` annotations have been omitted

```
class A {
  public B foo(D arg)  { .. }
}
class B extends A {
  public C foo(D arg) { .. }
  public A foo(A arg) { .. }
}
class C extends B {
  public B foo(A arg) { .. }
}
class D extends A {
  public A foo(C arg) { .. }
}
```

Additionally, you are given the following variable declarations

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

9. [**MRQ**] *(2 points)*Select ALL methods that can be annotated with `@Override` annotation.

   A. `public B foo(D arg)` in class `A`

   B. ***`public C foo(D arg)` in class `B`***

   C. `public A foo(A arg)` in class `B`

   D. ***`public B foo(A arg)` in class `C`***

   E. `public A foo(C arg)` in class `D`

   F. none, the `@Override` annotation cannot be added to any methods

Overriding happens when the method has (1) the same name, (2) has the same number of parameters, (3) has the same types of parameters, and (4) the return type may be the subclass of the return type of the overridden method. So, `C foo(D arg)` in class `B` overrides `B foo(D arg)` in class `A` and `B foo(A arg)` in class `C` overrides `A foo(A arg)` in class `B`.

10. **[MCQ]** *(2 points)* Consider the following code fragment

```
a.foo(b);
```

Which of the following `foo` method is invoked by the code fragment above?

A. `public B foo(D arg)` in class `A`

B. `public C foo(D arg)` in class `B`

C. `public A foo(A arg)` in class `B`

D. `public B foo(A arg)` in class `C`

E. `public A foo(C arg)` in class `D`

F. ***none, it cannot compile***

Class `A` only has `foo(D)` so it cannot accept `B`.

11. **[MCQ]** *(2 points)* Consider the following code fragment

```
c.foo(d);
```

Which of the following `foo` method is invoked by the code fragment above?

A. `public B foo(D arg)` in class `A`

B. ***`public C foo(D arg)` in class `B`***

C. `public A foo(A arg)` in class `B`

D. `public B foo(A arg)` in class `C`

E. `public A foo(C arg)` in class `D`

F. none, it cannot compile

Class `C` only has `foo(A)` and `foo(D)` that can accept the class `D`. The most specific is `foo(D)`. The lowest in hierarchy is in class `B`.

12. [**MCQ**] *(2 points)* Consider the following code fragment

```
d.foo(d);
```

Which of the following `foo` method is invoked by the code fragment above?

A. ***`public B foo(D arg)` in class `A`***

B. `public C foo(D arg)` in class `B`

C. `public A foo(A arg)` in class `B`

D. `public B foo(A arg)` in class `C`

E. `public A foo(C arg)` in class `D`

F. none, it cannot compile

Class `D` only has `foo(C)` and `foo(D)` that can accept the class `D`. The most specific is `foo(D)`. The lowest in hierarchy is in class `A`.

Section IV: Exceptions (4 points)

For this section, you are given the following exceptions

```
class ExcA extends RuntimeException {}
class ExcB extends ExcA {}
class ExcC extends ExcB {}
```

Additionally, you are given the following code fragment

```
public static void f() {
  try {
    // Line A
    g();
    // Line B
  } catch(ExcC e) {
    // Line C
  } catch(ExcA e) {
    // Line D
  } finally {
    // Line E
  }
}
public static void g() {
  h();
  // Line F
}
public static void h() {
  // Line G
  throw new ExcB();
  // Line H
}
```

13. [**MRQ**] *(2 points)* Consider the following code fragment

```
f();
```

Select ALL the lines that will be executed.

A. ***Line A***
B. Line B
C. Line C
D. ***Line D***
E. ***Line E***
F. Line F
G. ***Line G***
H. Line H

The sequence is:

(a) Line A
- `g();`
- `h();`

(b) Line G
- `throw new ExcB();`
- Skips Line H because exception is thrown
- Back to method `g()`, skips Line F because it is not catching
- Back to method `f()`, skips Line B because it is not catching
- Skipping Line C because it catches subtype of `ExcB`

(c) Line D because `ExcA` is supertype of `ExcB`

(d) Line E because `finally` is always executed

14. [**ESSAY**] *(2 points)* If we change the declaration of `ExcA` to extend `Exception` instead of `RuntimeException`, the following code fragment will not compile

```
public static void foo() {
  try {
    bar();
  } catch(ExcA e) {
    // do nothing
  } catch(ExcC e) {
```

```
    // do nothing
  }
}
public static void bar() {
  throw new ExcB();
}
```

Explain *in no more than three sentences* ALL the changes that you need to make to the code fragment above so that the code fragment will compile when we changed `ExcA` to extend `Exception` instead of `RuntimeException`.

Change the order of catch to catch `ExcC` before `ExcA` because `ExcC <: ExcA` and otherwise `ExcC` will never be caught and this causes compile error.
Add `throws Exception`, `throws ExcB`, or `throws ExcA` (*one of them is fine*) because now these are checked exceptions.

Section V: Types (14 points)

In some questions in this section, you are given a set of subtyping relationships and you need to come up with a *possible* object-oriented design for each of the set. However, there is a possibility that the set of subtyping relationships is **invalid**.
An example of a **valid** set of subtyping relationships is shown below.

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

A *possible* implementation is as follows

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

An example of an **invalid** set of subtyping relationships is shown below.

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

If the set is

- **Valid:** Answer True for the question and give a *possible* design.
- **Invalid:** answer False and briefly explain in one sentence why the set is impossible.

An obvious solution is to make everything `interface`. As the wording is unclear, we will accept this. Our solution will attempt to minimise the number of `interface`s. It is only invalid if there is a cycle on the subtyping relationship.

15. [**TRUE/FALSE**] *(3 points)* Is the following set of subtyping relationship valid? Give an example of a possible design for types A1, A2, A3, and A4 if the set is valid. Otherwise, briefly explain why the set is invalid.

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

No marks will be awarded if no design or explanation is given.

True.

```
interface A1 {}
interface A2 extends A1 {}
// A2 <: A1
class A4 {}
class A3 extends A4 implements A2 {}
// A3 <: A4  &  A3 <: A2  &  A3 <: A1
```

16. [**TRUE/FALSE**] *(3 points)* Is the following set of subtyping relationship valid? Give an example of a possible design for types A1, A2, A3, and A4 if the set is valid. Otherwise, briefly explain why the set is invalid.

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

No marks will be awarded if no design or explanation is given.

False. There is a cyclic subtyping: `A1 <: A2 <: A3 <: A4 <: A1`.

17. [**TRUE/FALSE**] *(3 points)* Is the following set of subtyping relationship valid? Give an example of a possible design for types A1, A2, A3, and A4 if the set is valid. Otherwise, briefly explain why the set is invalid.

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

No marks will be awarded if no design or explanation is given.

True.

```
interface A4 {}
class A1 implements A4 {}
// A1 <: A4
interface A3 extends A4  {}
// A3 <: A4
class A2 extends A1 implements A3  {}
// A2 <: A1  &  A2 <: A3
```

18. [**TRUE/FALSE**] *(3 points)* Is the following set of subtyping relationship valid? Give an example of a possible design for types A1, A2, A3, and A4 if the set is valid. Otherwise, briefly explain why the set is invalid.

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

No marks will be awarded if no design or explanation is given.

True.

```
interface A1 {}
class A2 implements A1 {}
// A2 <: A1
class A3 extends A2  {}
// A3 <: A2
class A4 extends A2 {}
// A4 <: A2  &  A4 <: A1
```

19. [**MRQ**] *(2 points)* Consider the following code fragment

```
Integer i = Integer.valueOf(3);
X x = i;
```

Select ALL valid options for the type `X`?

A. *`Object`*

B. *`int`*

C. `double`

D. `String`

`Object` is allowed due to subtyping relationship (*i.e.,* `Integer <: Number <: Object` *but you just need to remember* `Integer <: Object` *because* `Object` *is the root of the class hierarchy*). `int` is allowed due to auto-unboxing. But auto-unboxing cannot be automatically followed by subtyping. So even if `int <: double`, we cannot auto-unbox and then follow widening.

Section VI: Generics (12 points)

20. [**MCQ**] *(2 points)* Consider the following generic class

```
Integer i = Integer.valueOf(3);
class Tesla<T, S extends Car<T>> {
  T obj1;
  S obj2;
}
```

What will be the type of `obj1` after type erasure?

A. *`Object`*

B. `Car`

C. `Car<T>`

D. `Car<Object>`

21. [**MCQ**] *(2 points)* Consider the following generic class

```
Integer i = Integer.valueOf(3);
class Tesla<T, S extends Car<T>> {
  T obj1;
  S obj2;
}
```

What will be the type of `obj2` after type erasure?

A. `Object`

B. *`Car`*

C. `Car<T>`

D. `Car<Object>`

Type erasure removes type parameter completely so there is no `Car<T>` or `Car<Object>` but it will just be `Car`. Also, remember that `S` has an upper bound which is `Car<T>`.

22. [**TRUE/FALSE**] *(4 points)* The following code will compile without any syntax error or warning. True or False? Explain your reasoning in *at most four sentences*.

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

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

No marks will be awarded if no explanation is given.

True. Here, the problem is whether `S foo()` in `B` is an actual override or not. Remember, it can be an override as long as the return type is the subtype of `T` and we guarantee that `S` is indeed a subtype of `T` due to `S extends T`.

23. [**TRUE/FALSE**] *(4 points)* The following code will compile without any syntax error or warning. True or False? Explain your reasoning in *at most four sentences*.

```
class X {}
class Y extends X {}

class A<T> {
  public T foo(T a, X b) {
    return null;
  }
  public T foo(T a, Y b) {
    return null;
  }
}
```

No marks will be awarded if no explanation is given.

True. Here, the problem is whether `foo(T a, X b)` overloads `foo(T a, Y b)` or not. After type erasure, we have `foo(Object a, X b)` and `foo(Object a, Y b)` so it is a correct method overloading.

Section VII: OOP Principles (12 points)

For this section, consider the following Java program

```
class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  @Override public boolean equals(Object obj) {
    if (this == obj) { return true; }
    if (obj instanceof Point) {
      Point p = (Point) obj;
      return this.x == p.x && this.y == p.y;
    }
    return false;
  }
}

class Circle extends Point {
  private double r;
  public Circle(double x, double y, double r) {
    super(x, y);
    this.r = r;
  }
  @Override public boolean equals(Object obj) {
    if (this == obj) { return true; }
    if (obj instanceof Circle) {
      Circle c = (Circle) obj;
      return this.r == c.r && super.equals(c);
    }
    return false;
  }
}
```

An important property of the `equals` method as written in the Java documentation is reproduced here:

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

24. [**TRUE/FALSE**] *(4 points)* The program above violates information hiding. True or False? Answer True if the program violates information hiding and answer False if the program does not violate information hiding.

    Explain your reasoning in no more than two sentences with reference to the program above. No marks will be awarded if no explanation is given.

False. All access modifiers are `private` and there is no getter or setter for both `Point` or `Circle`.

25. [**TRUE/FALSE**] *(4 points)* The program above violates LSP. True or False? Answer True if the program violates LSP and answer False if the program does not violate LSP.

    Explain your reasoning in no more than two sentences with reference to the program above. No marks will be awarded if no explanation is given.

True. The reflexivity is violated because `new Point(0,0).equals(new Circle(0,0,1))` evaluates to `true` BUT `new Circle(0,0,1).equals(new Point(0,0))` evaluates to `false` because `Point` is not an instance of `Circle`. So, the `equals` method is no longer symmetric.

26. [**TRUE/FALSE**] *(4 points)* The program above violates the principle of "Tell, Don't Ask". True or False? Answer True if the program violates "Tell, Don't Ask" and answer False if the program does not violate "Tell, Don't Ask".

    Explain your reasoning in no more than two sentences with reference to the program above. No marks will be awarded if no explanation is given.

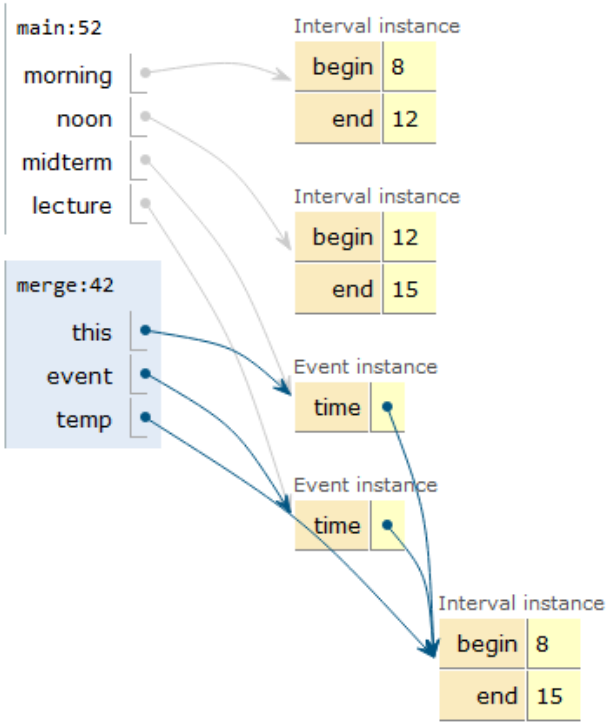
False. A possible violation is in `Circle::equals` where we try to access `x` and `y` directly. However, we use `super.equals(c)` instead so there is no violation of "Tell, Don't Ask" principle as we tell the superclass to check for equality directly rather than asking for the attributes/fields.

Section VIII: Stack and Heap (10 points)

Consider the following complete program

```
class Interval {
  private int begin;
  private int end;
  public Interval(int begin, int end) {
    this.begin = begin;
    this.end = end;
  }
  public Interval union(Interval time) {
    return new Interval(this.begin, time.end);
    // simplified as min/max does nothing for this particular case
  }
}
class Event {
  private Interval time;
  public Event(Interval time) {
    this.time = time;
  }
  public void merge(Event event) {
    Interval temp = this.time.union(event.time);
    this.time = temp;
    event.time = temp;
    // Line A
  }
}
public class StudentLife {
  public static void main(String[] args) {
    Interval morning = new Interval(8, 12);
    Interval noon = new Interval(12, 15);
    Event midterm = new Event(morning);
    Event lecture = new Event(noon);
    midterm.merge(lecture);
  }
}
```

27. Draw the stack and heap diagram on a piece of paper when the program executes the main method in `StudentLife` class up to "**Line A**". Do NOT remove any objects created on the heap. Label your stack with the method name as shown in recitation. You may ignore any variables used in `main` but not shown in the program.



1. `morning` points to an `Interval` instance with `begin = 8`, `end = 12`.
2. `noon` points to an `Interval` instance with `begin = 12`, `end = 15`.
3. `midterm` points to an `Event` instance.
4. `lecture` points to an `Event` instance.
5. `temp` points to an `Interval` instance with `begin = 8`, `end = 15`.
6. `this.time` and `event.time` alias.
7. `this` and `midterm` alias.
8. `event` and `lecture` alias.
9. `StudentLife::main` has `morning`, `noon`, `midterm`, and `lecture`.
10. `Event::merge` has `this`, `event`, and `temp`.

END OF PAPER