CS2030S: Programming Methodology II
AY 2022-2023 – Semester 1

Midterm Exam

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. Write your matric number below

A								

Marks

Section	**Subtotal**
Java Basic	6
Interface and Abstract Class	4
Inheritance, Polymorphism, and Dynamic Binding	8
Exceptions	4
Types	14
Generics	12
OOP Principles	12
Stack and Heap	10
Total	**70**

Good Luck!

Section I: Java Basic *(6 Points)*

1. Select ALL statements that are **true** about Java type systems.

   (✓) Java is statically typed
   (B) Java is dynamically typed
   (✓) Java is strongly typed
   (D) Java is weakly typed

   Answer: 1 *points*

2. Select ALL constructs that are resolved via *static binding* instead of dynamic binding.

   (A) Non-static method
   (✓) Static method
   (✓) Non-static field
   (✓) Static field

   Answer: 1 *points*

3. Select ALL statements that are **false** about Java `final` keyword.

   (✓) `final` keyword can be used to prevent a method from being inherited
   (✓) `final` keyword can be used to prevent a method from being overloaded
   (C) `final` keyword can be used to prevent a method from being overridden
   (D) `final` keyword can be used to prevent a class from being inherited

   Answer: (A) is an answer because it's a (*i.e., only one*) method 1 *points*

4. 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
   (✓) Method overloading
   (C) Inheritance
   (D) Polymorphism

   Answer: No polymorphism because no inheritance. 1 *points*

5. Select ALL statements that are **true** about *complex types* in Java.

(A) Java generic is covariant	(✓) Java array is covariant
(B) Java generic is contravariant	(E) Java array is contravariant
(✓) Java generic is invariant	(F) Java array is invariant

Answer: □ *1 points*

6. Select ALL the statements that are **true** regarding *access modifier* in Java.

(✓) 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

(✓) Fields/methods with `public` access modifier can be accessed by code inside the class

(✓) Fields/methods with `public` access modifier can be accessed by code in the subclass

Answer: □ *1 points*

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. Select ALL classes that *must* have the `abstract` modifier in the class declaration.

   (✓) Class `C1`

   (✓) Class `C2`

   (✓) Class `C3`

   (D) Class `C4`

   Answer: Simply check which classes still have *abstract* methods. *2 points*

8. 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`

   (✓) `C3` <: `I1`

   (✓) `C4` <: `I2`

   Answer: *2 points*

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. Select ALL methods that can be annotated with `@Override` annotation.

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

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

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

(✓) `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

Answer: *2 points*

10. 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`

(✓) none, it cannot compile

Answer: *2 points*

11. Consider the following code fragment

```
c.foo(b); // originally 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`

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

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

(F) none, it cannot compile

Answer: If it is `c.foo(d)`, then the answer is B. *2 points*

12. Consider the following code fragment

```
d.foo(b); // originally, 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`

(✓) none, it cannot compile

Answer: If it is `d.foo(d)`, then the answer is A. *2 points*

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. Consider the following code fragment

```
f();
```

Select ALL the lines that will be executed.

(✓) Line A

(B) Line B

(C) Line C

(✓) Line D

(✓) Line E

(F) Line F

(✓) Line G

(H) Line H

Answer: The sequence is `A` → `G` → `D` → `E`.

2 points

14. If we change the declaration of `ExcA` to extend `Exception` instead of `RuntimeException`, the following code fragment will not compile

```java
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`.

Answer: *2 points*

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` (*alternatively,* `throws ExcB` or `throws ExcA`) 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 such that:

- All the explicitly stated subtyping relationship are satisfied.
- All the implicitly (*i.e., derived*) stated subtyping relationship are satisfied.
- No additional subtyping relationship not explicitly/implicitly stated.
- Only a *minimal* number of `interface` can be used.

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. In your design, you must minimise the number of interfaces.
- **Invalid:** Answer `false` and briefly explain in one sentence why the set is impossible.

15. 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.

Answer: *3 points*

True/~~False~~

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

16. 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.

Answer: *3 points*

~~True~~/False

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

17. 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.

Answer: *3 points*

True/~~False~~

```
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. 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.

Answer: *3 points*

True/~~False~~

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

19. Consider the following code fragment

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

Select ALL valid options for the type `X`?

(✓) `Object`

(✓) `int`

(✓) `double`

(D) `String`

Answer: `double` is allowed because *unboxing* then *widening*. *2 points*

Section VI: Generics (12 points)

20. 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?

(✓) `Object`

(B) `Car`

(C) `Car<T>`

(D) `Car<Object>`

Answer: *2 points*

21. 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`

(✓) `Car`

(C) `Car<T>`

(D) `Car<Object>`

Answer: *2 points*

22. 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.

Answer: *4 points*

True/~~False~~

Here, the question is whether `S foo()` in B is a *valid override* or not. Recap that it can be an override as long as the *return type is the subtype* of `T`. Here, we guarantee that `S` is the subtype of `T` because we declare the type parameter as `S extends T`.

23. 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.

Answer: *4 points*

True/~~False~~

Here, the question 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)` as the *method signature* because both `X` and `Y` are not type parameter. As such, the method overloading is allowed.

Section VII: OOP Principles (12 points)

For this section, consider the following Java program

```java
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. 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.

    Answer: *4 points*

    ~~True~~/False

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

25. 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.

    Answer: *4 points*

    True/~~False~~

    The *symmetry* property of `equals` is violated because if we let `p1 = new Point(0,0)` and `p2 = new Circle(0,0,1)` then `p1.equals(p2)` is `true` but `p2.equals(p1)` is `false`.

26. 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.

Answer: *4 points*

~~True~~/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 because we *tell* the superclass to check instead of *asking* for the value of `x` or `y`.

Section VIII: Stack and Heap (10 points)

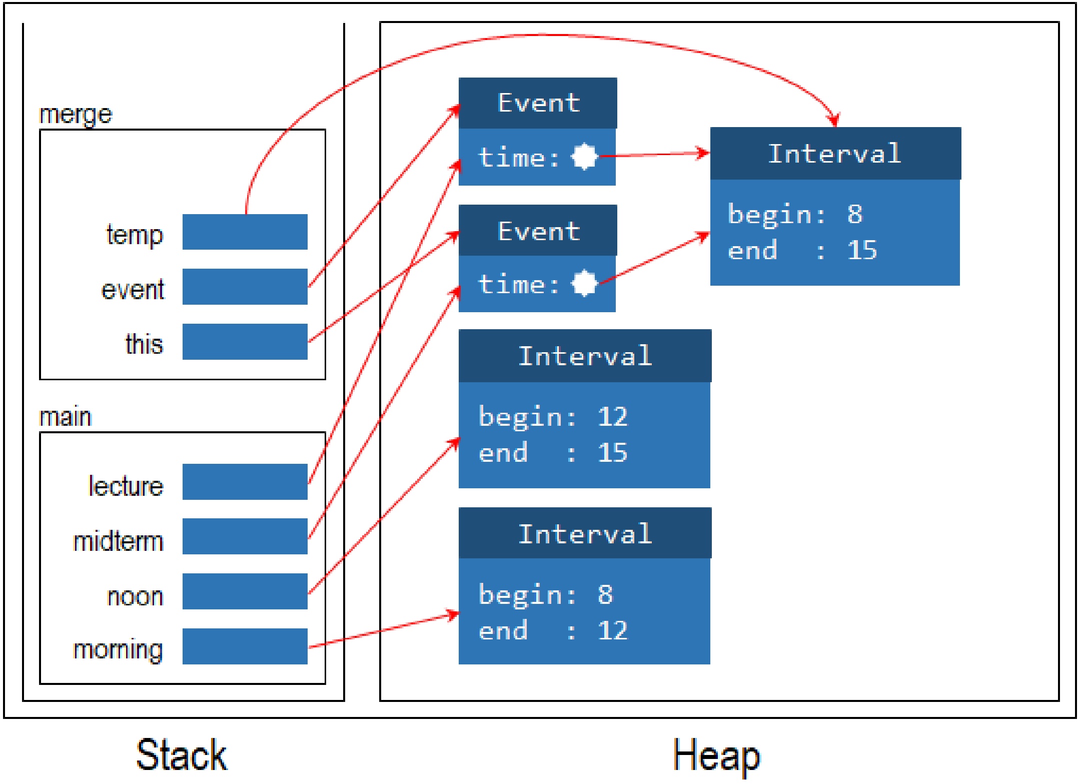
Consider the following complete program

```
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);
  }
}

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.

Answer: *10 points*

You may expand the template shown below.



Check:

1. `morning` points to an `Interval` instance with `begin = 8` and `end = 12`.
2. `noon` points to an `Interval` instance with `begin = 12` and `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` and `end = 15`.
6. `this.time` and `event.time` are *alias*.
7. `this` and `midterm` are *alias*.
8. `midterm` and `lecture` are *alias*.
9. The *frame* of `StudentLife::main` contains: `morning`, `noon`, `midterm`, and `lecture`.
10. The *frame* of `Event::merge` contains: `this`, `event`, and `temp`.

The following page is intentionally left blank

End of Paper