NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
MIDTERM ASSESSMENT FOR
Semester 2 AY2017/2018

CS2030 Programming Methodology II

March 2018 Time Allowed 90 Minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 13 questions and comprises 12 printed pages, including this page.
2. A 4-page answer sheet is also given. Write all your answers in the answer sheet. Submit your answer sheet at the end of the assessment.
3. The total marks for this assessment is 50. Answer **ALL** questions.
4. This is an **OPEN BOOK** assessment.
5. All questions in this assessment paper use Java 8 unless specified otherwise.
6. State any additional assumption that you make.

Part I

Multiple Choice Questions (24 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 statements about inheritance in Java is FALSE?
   A. We can use the `extends` keyword to specify inheritance
   B. A class can `extends` from at most one other class
   C. A class declared as `final` cannot be inherited
   D. A method declared as `final` cannot be overridden
   E. A field declared as `final` cannot be accessed by the subclass

   Write `X` in the answer box if none of the statements above is false.

   Solution: E. A `final` field cannot be modified after initialization, whether it can be accessed or not, depends on its access modifier.

2. (3 points) Recall that we can override the method `equals` in the class `Point` (as defined in CS2030) so that two `Point` objects are equal if they have the same `x` and `y` coordinates.

   Overriding `equals` may or may not violate the Liskov Substitution Principle (LSP). It depends on what the specified properties of `equals` in the class `Object` are.

   Which of the following property of `equals`, if specified, would cause `Point` to violate the LSP?

   For two variables of type `Object`, `o1` and `o2`, `o1.equals(o2)` is true if and only if

   (i) `o1 == o2`
   (ii) `o2.equals(o1)` is also true
   (iii) `o1.equals(o3)` implies `o3.equals(o2)` for another variable `o3`.
   A. (i) only
   B. (ii) only
   C. (i) and (ii) only
   D. (ii) and (iii) only

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

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

Solution: A. The `equals` method for `Point` does not satisfy (i). So if (i) is specified as a property of `equals`, `Point` would violate LSP.

3. (3 points) Consider the definition of `I`, `J`, `A`, and `B` below. In order for `B` to be a concrete (non-abstract) class, what methods should `B` implements?

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

interface J extends I {
  void g();
}

abstract class A implements J {
  public void g(int x) {
    return;
  }

  abstract public void h();
}

class B extends A {
  :
}
```

   A. `h` only
   B. `f` and `h` only
   C. `f` and `g` only
   D. `g` and `h` only
   E. `f`, `g` and `h`

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

Solution: E. `f` and `g` should be obvious. There are two methods named `g` but their signature is different. So, we still need to override `g`.

4. (3 points) Consider the definition of classes `A`, `B`, and `C` below.

```
class A {
  void f(int x) {
    System.out.println("A");
  }
}

class B extends A {
  void f(int x) {
    System.out.println("B");
  }
}

class C extends A {
  void f(String x) {
    System.out.println("C");
  }
}
```

Which of the following declaration and initialization of variable `x` would cause `x.f(1)` to print the string "A"?

(i) `A x = new B();`

(ii) `A x = new C();`

(iii) `B x = new B();`

(iv) `C x = new C();`

A. (ii) only

B. (ii) and (iv) only

C. (i) and (iii) only

D. (i), (ii), and (iii) only

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

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

Solution: B. The method chosen during compile time must have the signature `void f(int i)`. Regardless of whether the compile-time type of `x` is `A`, `B`, or `C`, this signature will be chosen.

During run-time, we want `A::f` to be called. So the run-time type of `x` cannot be `B`, as this would call `B::f` instead. `x` must have the run-time type of either `A` or `C`.

5. (3 points) **Note: Nested class is not within the scope of midterm after AY21/22.**

Consider the following class `Out` which contains an inner class `In` and a local class `Local`

```
class Out {
  int x;

  class In {
    int y;
  }

  void foo(int z) {
    x = 1; // (A)
    z = 1; // (B)
    class Local extends In {
      void bar() {
        int w;
        w = x; // (C)
        w = y; // (D)
        w = z; // (E)
      }
    }
  }
}
```

Which of the following statement about the five statements labeled (A)-(E) above is FALSE:

A. Statement (A) causes a compilation error, as Java does not allow the value of `x` to be changed inside the method `foo` if `x` is captured by `Local`.

B. Statement (B) causes a compilation error, as Java does not allow the value of `z` to be changed inside the method `foo` if `z` is captured by `Local`.

C. Statement (C) compiles without error, as the method `bar` can access the field `x`.

D. Statement (D) compiles without error, as the method `bar` can access the field `y`

E. Statement (E) causes a compilation error, as Java does not allow variable capture of `z`, which is neither final or effectively final.

Write `X` in the answer box if none of the statements above is false.

Solution: A. x is not a local variable, so accessing x from within Local is OK.

6. (3 points) Suppose we have three types S, T, and U, with the following subtype relationship

$$U <: T <: S$$

Let $A(X)$ be a complex type that depends on type X.

Which of the following statement is FALSE:

A. Assigning a variable of type U to a variable of type S is a form of widening type conversion.

B. Assigning a variable of type S to a variable of type T requires type casting in Java.

C. We can pass a variable of type S to a method expecting type T as an argument without type casting.

D. Passing a variable of type T to a method expecting an argument of type S will never raise a runtime `ClassCastException`.

E. If $A(T) <: A(S)$, then we say that A is covariant

Write X in the answer box if none of the statements above is false.

Solution: C. Passing S to T is a narrowing conversion, so type casting is needed.

7. (3 points) Suppose we have a generic class with two type parameters:

```
class Pair<T, U> {
  T first;
  U second;
}
```

Which of the following code will lead to a compilation error?

(i) `Pair<String, String> p = new Pair<>();`

(ii) `Pair<int, int> p = new Pair<>();`

(iii) `Pair<> p = new Pair<Object, Object>();`

(iv) `Pair<?, ?> p = new Pair<String, Object>();`

A. (ii) only

B. (i) and (iv) only

C. (ii) and (iii) only

D. (i), (iii), and (iv) only

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

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

Solution: C. (ii) won't work since it uses primitive types. (iii) won't work since <> operator can be used only when we instantiate a generic type.

8. (3 points) Consider the code below. `InterruptedException` is a subclass of `Exception`.

```
class Inception {
  public static void main(String args[]) {
    van();
  }

  static void van() {
    try {
      System.out.println("van");
      hotel();
    } catch (Exception e) {
      System.out.println("exception (van)");
    }
  }

  static void hotel() throws InterruptedException {
    try {
      System.out.println("hotel");
      snowFortress();
    } catch (Exception e) {
      System.out.println("exception (hotel)");
    }
  }

  static void snowFortress() throws InterruptedException {
    System.out.println("snow fortress");
    limbo();
  }

  static void limbo() throws InterruptedException {
    throw new InterruptedException();
  }
}
```

Which of the following string will NOT be printed when we invoke the main class `Inception`?

A. `van`

B. `hotel`

C. `snow fortress`

D. `exception (van)`

E. `exception (hotel)`

Write `X` in the answer box if every string above is printed.

Solution: D.

Part II

Short Questions (24 points)

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

9. (4 points) **Modeling**

Suppose you want to model the following scenario in an object-oriented program.

A module has multiple assessments. There are three types of assessments: lab assignment, test, and project, each to be graded differently.

(a) (3 points) List down the name of the five classes, and the relationship (either IS-A or HAS-A) between them.

(b) (1 point) Identify an opportunity to use polymorphism in the scenario above.

Note: you do not have to write any Java code to answer this question.

Solution: Module, Assessment, Lab, Test, Project. Module HAS-A Assessment. Each of Lab, Test, Project IS-A Assessment. The method `grade` in Assessment can be overridden by individual sub-classes – polymorphism can be used here.

This should be quite straightforward. Some common mistakes include introducing new classes not mentioned in the scenario and missing out the more obvious classes. (e.g., Grade, GradeBook, etc).

Some students also wrote the IS-A relationship in the wrong direction (Assessment IS-A Lab).

Some students give vague answers when asked to identify the opportunity for polymorphism (e.g., inheritance is an opportunity for polymorphism).

10. (3 points) **Hash Code.**

Note: Hash code is not within the scope of midterm after AY21/22.

Recall that whenever we override the method `equals()` from the class `Object`, we must override the method `hashCode()` as well. It is required that two objects `x` and `y` satisfy the following property P:

if `x.equals(y)`, then `x.hashCode() == y.hashCode()`

Someone presented to you the following implementation of `hashCode()` for the class `A`. The other parts of class `A` are omitted (including implementation of `equals()`).

```
class A {
    :
  @Override
  int hashCode() {
    return 8888;
```

```
    }
}
```

(a) (1 point) Does the implementation of `hashCode()` above satisfy property P?

(b) (2 points) The implementation of `hashCode()` above is, however, considered a bad practice. Why?

Solution:

(a) Yes.

(b) (i) All elements will be hashed to the same bucket in `HashSet` and `HashMap`, so searching and retriving will be inefficient as we have to search through all elements everytime. (ii) We cannot use `hashCode()` to filter out two objects that are different in `equals`.

If you explain either one of the above, you get 2 marks.

If you mention something along the line, but is vague (e.g., "cause some trouble with `HashSet`", "voilates the expectation of hashing"), you get 1 mark.

Many students assume that if `hashCode()` returns the same value, implies that `equals()` returns true. Answers that say this or "we can only have one item in `HashSet`", "`HashMap` will stop working", "every instance of A would be equal", etc. will not get any marks.

Another common wrong answer is that the new `hashCode()` violates the Liskov Substitution Principle (LSP). To say that something violates LSP, we must be clear about what is the desirable property of `hashCode`. The only property is P, which Part (a) already establishes that it is not violated.

Another common misconception is that different objects of the same class *must* give a different hash code. This is not true. Since `hashCode` returns an int, there are only 2^32 possible hash code. Take strings for instance, there are many many more stirngs than 2^32. So some strings must give us the same hash code. A concrete example: `Arrays.hashCode`. Both {0, 1} and {1, -30} gives the same hash code.

11. (8 points) **Method Overriding.**

During the lectures, we have seen that, if we have two methods with the same method signature, one in the superclass and the other in a subclass, then the method in the subclass will override the method in the superclass. We, however, did not say much about the return type of the overridden and the overriding methods. We will explore more about that in this question.

Let's construct a simple example. Suppose we have two classes, class `A` and class `B` inherits from `A`. Both classes `A` and `B` define a method `A copy()`, as seen below, that returns a copy of the object.

```
class A {
  int x;

  A(int x) {
    this.x = x;
  }

  public A copy() {
    return new A(x);
  }
}

class B extends A {
  int y;

  B(int x, int y) {
    super(x);
    this.y = y;
  }

  @Override
  public A copy() { // Line 22
    return new B(x, y);
  }
}
```

(a) (2 points) Why does the following code gives a compilation error? Fix the code below so that the compilation error goes away.

```
B b1 = new B(1, 2);
B b2 = b1.copy();
```

> **Solution:** The return type of b1.copy() is A. Assigning the return value to b2 is a narrowing conversion and needs type casting. B b2 = (B) b1.copy();

(b) (2 points) Which version of `copy()` will the line `a1.copy()` below invoke? The one in class `A`, or in `B`?

```
A a1 = new B(1, 2);
A a2 = a1.copy();
```

Solution: B

(c) (4 points) Suppose we change Line 22 above so that the return type of method `copy()` is `B` instead of `A`. Java compiler does not give any compilation error and allows `copy()` in class `B` to override `copy()` in class `A`. Explain why it is safe for Java to allow this.

Solution: Existing code that has been written to invoke A's `copy` would still work if the code invokes B's `copy` instead after `B` inherits from `A`.

The following answers are insufficient / wrong.

- `B` is subtype of `A` so it is OK. This does not explain in the context of the return type in an overriding method. For instance, Java does not allow `foo(B x)` to override `foo(A x)`. But it is close since subtyping implies that code written expecting objects of type A can be used with objects of type B. (2 marks) However, if you elaborated on subtyping of B from A to any of the following, then you are not applying the concept of subtyping correctly to answer this question.
- `B`'s `copy()` returns an object of type `B`, so it is safe to change the return type to `B`. This does not answer the question, which is why Java allows overriding (not why Java allows the return type to be `B`, which is rather obvious). (0 marks)
- It does not violate LSP. `B`'s `copy` still returns a copy of the object. This has to do with the semantics of the program. Java's compiler does not check for the semantics and violation of LSP. (0 marks).
- There is no ambiguity as to which version of `copy` will be invoked; Or, the return type is not part of a method signature. This does not explain why Java does not allow, say, `int copy() B` to override `A copy()` in `A`. (0 marks)
- Assigning `B` to `A` is a widening conversion so it is OK. Again, this does not explain why Java allows overriding. (0 marks).

12. (4 points) **Type.**

You are shown the implementation of a class with the following two methods.

```
void printPositiveBytesFromIntegers(List<Integer> list) {
  for (Integer i : list) {
    if (i.byteValue() > 0) {
      System.out.println(i.byteValue());
    }
  }
}

void printPositiveBytesFromLong(List<Long> list) {
  for (Long i : list) {
    if (i.byteValue() > 0) {
      System.out.println(i.byteValue());
    }
  }
}
```

The methods go through, a list of `Integer` objects and a list of `Long` objects, round or truncate them to a value of type `byte`, and print out the value if it is positive. You are asked to copy-and-paste the methods given and change them to produce methods that perform the same action but on a list of other types. One for a list of `Double` objects, one for a list of `Short` objects, one for a list of `Float` objects, etc.

You recall the abstraction principle from CS2030, and you know that copying-and-pasting the code multiple times is not the best way to do this. You look up the Java API, and found that:

- `Integer`, `Long`, `Double`, `Short`, and `Float` are all subclasses of the abstract class `Number`.
- `byteValue()` is a non-abstract method defined in the class `Number` and it does exactly what the code above intended it to do.

With this information, and with what you learn about generic types, you are now ready to write only ONE method to replace the five methods that would have been produced if you naively replicate the methods, one for each type. Your method should be able to take in a list of type `List<Integer>`, `List<Long>`, `List<Double>`, `List<Short>`, or `List<Float>` as an argument. In fact, your method is so general that a list of any subtype of `Number` can be passed in as an argument.

Write this method in the space given on the answer sheet.

Solution:

```
void printPositiveBytes(List<? extends Number> list) {  // Line A
  for (Number i : list) { // Line B
    if (i.byteValue() > 0) {
      System.out.println(i.byteValue());
    }
```

```
  }
}

<T extends Number> void printPositiveBytess(List<T> list) { // Line A
  for (Number i : list) {  // Line B
    if (i.byteValue() > 0) {
      System.out.println(i.byteValue());
    }
  }
}
```

You get 2 marks for choosing the correct type for `list` in Line A and 2 marks for the correct type of `i` in Line B.

Some common mistakes:

- Using `T` without `<T>` to make the method generic;
- Mix the use of `T` and `?`;
- Use `List<Number>` as the type for `list`;
- Use `Object`, `?`, or `<? extends Number>` as the type for `i`.

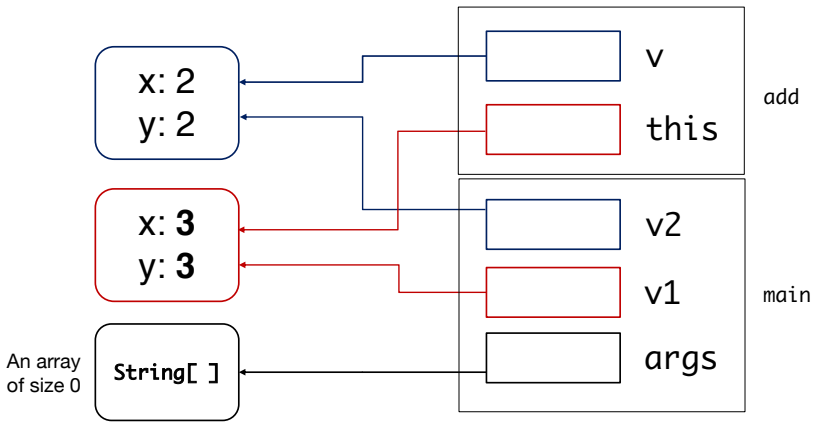
13. (5 points) **Heap and Stack.**

Consider the following definition of a `Vector2D` class:

```
class Vector2D {
  private double x;
  private double y;

  Vector2D(double x, double y) {
    this.x = x;
    this.y = y;
  }

  void add(Vector2D v) {
    this.x += v.x;
    this.y += v.y;
    // line A
  }
}

class Main {
  public static void main(String[] args) {
    Vector2D v1 = new Vector2D(1, 1);
    Vector2D v2 = new Vector2D(2, 2);
    v1.add(v2);
  }
}
```

We execute the `Main` class without any command line argument. Show the content on the stack and the heap when the execution reaches the line labeled A above. Label your variables and the values they hold clearly. You can use arrows to indicate object references. Draw boxes around the stack frames of the methods `main` and `add` and label them.

Solution:



Common mistakes include:

- Forgetting the `args` is a method parameter to main so should be allocated on the stack. Java's convention is that `args` points to an empty array, but we are fine with `args` pointing to `null` too.
- Did not update the value of `v1` to (3, 3).
- Give the wrong order of stack frame.

END OF PAPER

This page is intentionally left blank.