The University of Melbourne
School of Computing and Information Systems

Semester 2, 2018 Sample Final Exam

COMP90041
Programming and Software Development

[redacted] Sample Answers Included [redacted]

Reading Time: 15 minutes
Writing Time: 2 hours
This paper has 9 pages.

Total marks for this paper: 60

Authorised Materials:

Writing instruments (e.g., pens, pencils, erasers, rulers).
No other materials and no electronic devices are permitted.

Instructions to Invigilators:

The exam paper must remain in the exam room and be returned to the subject coordinator.

Instructions to Students:

The marks for each question are listed at the beginning of the question. You should attempt all questions. Use the number of marks allocated to a question as a rough indication of the time to spend on it. We have tried to provide ample space for your answers; do not take the amount of space provided for an answer as an indication of how much you need to write.

This paper must *not* be lodged with the university library.

This page intentionally left blank

Question 1 [6 marks]

Consider a method whose definition is the following:

```
static String testmethod(int n)
{
     String r = "none";

     switch (n)
     {

     case 1: r = "one";
     case 2: r = "two";
     case 3: r = "three";

     }

     return r;
}
```

What string is returned by each of the following calls?

(a) `testmethod(1)`
`"three"` ⇐

(b) `testmethod(2)`
`"three"` ⇐

(c) `testmethod(8)`
`"none"` ⇐

Question 2 [6 marks]

There are some actions that the implementations of static methods cannot perform that the implementations of non-static methods can perform. Give an example, and give the reason why Java does not allow static methods to perform that action.

Sample Answer to Question 2 ⇐

Static methods cannot reference the `this` implicit parameter, because static methods can be, and usually are, invoked without sending a message to any object. The lack of `this` implicit parameter prevents many other actions that can be performed by non-static methods:

- accessing instance variables of the class

- setting instance variables of the class
- calling non-static methods of the class

All of these things implicitly use the `this` implicit parameter.

The following code shows examples of forbidden actions:

```
public class Foo {
    int instanceVar;
    public int cantUse() {
        return instanceVar;
    }

    public static void badMethod(int n) {
        int local = instanceVar; // forbidden
        instanceVar = 7; // forbidden
        return cantUse(); // forbidden
    }
}
```

Question 3 **[6 marks]**

As of Java 1.5, Java supports generic types, for example `ArrayList`. What is a generic type? How is this an improvement on the `ArrayList` class of Java 1.4, when Java did not support generics?

⇒**Sample Answer to Question 3**

A *generic* type is one that requires one or more parameter types before they specify a type. For example, a variable may be declared to be `ArrayList`⟨`String`⟩, indicating that its elements are all strings, while the elements of an `ArrayList`⟨`Integer`⟩ are all integers.

Generics permit the programmer to better specify their intentions, and allow the Java compiler to produce better error messages when the intentions are violated. Prior to the introduction of generics in Java 5, it was not possible for the programmer to prevent objects of any type from being stored in an `ArrayList`; every object in an `ArrayList` could be a different type. Furthermore, each object taken from an `ArrayList` needed to be cast to the appropriate type, and the cast would fail at runtime if the object was not the correct type.

As of Java 5, the type of object in an `ArrayList` can be specified, and the compiler will ensure that only objects of that that type can be stored in the `ArrayList`. There is also no need to cast objects taken from the `ArrayList`; since only one type of object can be stored in the `ArrayList`, only that type will come out of it.

```
// Prior to Java 5:
ArrayList a = new ArrayList();
a.add("hello");
...
String s = (String)a.get(0); // cast could fail

// Since Java 5:
ArrayList<String> a = new ArrayList<String>();
a.add("hello");
...
String s = a.get(0); // no cast needed
```

Question 4 [3 marks]

What will this code fragment print?

```
int x=3, y=0;
while (x>=0) {
     y++;
     x--;
}
System.out.println(y);
```

Sample Answer to Question 4 ⇐

```
4
```

Question 5 [3 marks]

What will this code fragment print?

```
int[] a = {1,1,2};
int sum = 0;
for (int i=1; i<=3; ++i) {
     sum += a[i];
}
```

```
        System.out.println(sum);
```

⇛**Sample Answer to Question 5**

There's a runtime error (index out of bounds).

Question 6 **[6 marks]**

A *privacy leak* in a Java program occurs when a class's internal data can be manipulated by methods of other classes, despite being declared `private`. List at least two ways this can happen, and give an example. List as many ways as you can think of for the author of the class to prevent privacy leaks.

⇛**Sample Answer to Question 6**

A privacy leak happens when a method has access to the private internal state of another class. This happens when a class `C`'s method returns a mutable object (or array) stored in one of `C`'s private instance variables, or when it stores a mutable object obtained from another method in one of `C`'s private instance variables, or when it passes a mutable object stored in `C`'s private instance variable as argument to a method of another object. In all of these cases, a method of another class could alter the state of an instance of `C`, circumvent its `private` declaration and any careful controls over modifications made to instances of `C` simply by modifying the object stored in `C`'s instance variable.

Privacy leaks can be avoided in any of the following ways:

- Use an immutable object, such as a `String` to hold the data. This is safe since no method may modify it.
- Copy (clone) any object to be stored in `C`'s instance variables, and copy any object from `C`'s instance variables to be returned to or passed to a method from another class.
- Simply do not have any methods that store such objects in `C`'s instance variables or return such objects to other methods. For example, instead of having an accessor to get a Person object's address as an array, have an accessor that gets the lines one at a time (if these are strings, they are immutable, so this is safe). And instead of having a mutator take an array of address lines as input, have it take the lines one at a time.

Question 7 **[6 marks]**

The println method of the System.out object can be used to print any object, regardless of which primitive type or class it belongs to. Outline the mechanism that println uses to accomplish this task.

Sample Answer to Question 7 ⇐

The System class has an instance variable named `out` whose class defines the `println` method. So when you write System.out.println(...), you are sending a println message to the System.out object. That message is heavily overloaded to work on all the primitive types, plus String and Object (you can see this in the documentation for the PrintStream class). If you pass it an object other than a string, it uses that object's `toString` method, possibly inherited from Object, but preferably overridden by the class in question, to produce a string representation of the object. The `println` method then sends that string to the output stream.

Question 8 **[9 marks]**

What is the difference between abstract and concrete (non-abstract) classes? What can you do with one that you cannot do with the other? Give an example of a situation in which you would use an abstract class, and explain why it would be appropriate in that situation.

Sample Answer to Question 8 ⇐

In Java, an *abstract* class is a class that is permitted to have abstract methods. non-abstract classes are not permitted to have abstract methods. An abstract method is a method with a complete signature (fully specified argument and return types), but in place of the method body, it has only a semicolon. Abstract methods are used to specify an interface without an implementation for certain methods. Because an abstract class is permitted to have methods with no implementation, it is not permitted to create instances of an abstract class. However, an abstract class can still be used as a type. Subclasses of an abstract class may override its abstract methods with concrete methods, thereby providing the implementation of the interface specified by the abstract class.

For example, an employee may be salaried or paid an hourly wage. It is not possible to say how much to pay an employee without knowing whether she is salaried or hourly. Thus is makes sense to have `SalariedEmployee` and `HourlyEmployee` classes, each of which specifies how to calculate that employee's pay. But by creating an abstract `Employee` class with an abstract `calculatePay` method, we create a type that describes both kinds of employee, so we can have a variable `staff` of type `Employee[]` that intermixes employees of both sorts, and can iterate over it calculating each employee's pay in the appropriate way.

Question 9 [15 marks]

Write two classes, `Position` and `Displacement`. A `Position` represents a Cartesian (x, y) position pair, and a `Displacement` represents a Cartesian distance, that is, a $(\delta x, \delta y)$ pair. Ensure that both classes are **immutable**. In both cases, values should be represented as `doubles`.

These classes should implement the following operations:

- Construct new `Position` and `Displacement` objects;
- Subtract one `Position` from another to get a `Displacement`;
- Add a `Displacement` to a `Position` to get a `Position`;
- Add two `Displacements` to get a `Displacement`;
- Scale (multiply) a `Displacement` by a scalar (`double`);
- Get the `x` and `y` components of both `Positions` and `Displacements`.

⇒Sample Answer to Question 9

Position.java:

```
public class Position {
    private final double x, y;
    public Position(double x, double y) {
        this.x = x; this.y = y;
    }

    public double getX() {return x;}
    public double getY() {return y;}

    public Displacement difference(Position other) {
        return new Displacement(x - other.x, y - other.y);
    }

    public Position addDisplacement(Displacement disp) {
        return new Position(x + disp.getDeltaX(), y + disp.getDeltaY());
    }
}
```

Displacement.java:

```
public class Displacement {
    private final double deltaX, deltaY;
    public Displacement(double deltaX, double deltaY) {
```

```
        this.deltaX = deltaX; this.deltaY = deltaY;
    }

    public double getDeltaX() {return deltaX;}
    public double getDeltaY() {return deltaY;}

    public Displacement add(Displacement other) {
        return new Displacement(deltaX + other.deltaX, deltaY + other.deltaY);
    }

    public Displacement scale(double factor) {
        return new Displacement(deltaX * factor, deltaY * factor);
    }
}
```

— End of Exam —