The University of Melbourne
Department of Computing and Information Systems

Semester 2, 2014 Sample Assessment

COMP90041 Programming and Software Development

Sample Answers Included

Reading Time: 15 minutes

Total marks for this paper: 60

Writing Time: 2 hours

This paper has 7 pages.

Authorised materials:

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

Instructions to Invigilators:

Supply students with standard script books.

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

Instructions to Students:

You should attempt all questions. You should use the number of marks allocated to a question to judge the level of detail required in your answer.

This paper should *not* be lodged with Baillieu Library.

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?

1. testmethod(1)
2. testmethod(2)
3. testmethod(8)

Sample Answer to Question 1

1. `"three"`
2. `"three"`
3. `"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]**

What can you do with a class member of public visibility that you cannot do with a class member of protected visibility?

Sample Answer to Question 3

You cannot access members of protected visibility from classes in a different package that are not descendent classes of the class holding the protected member. Public members can be accessed from any class. For example, if a class `A` defines a protected member `m` and public member `p`, then `m` can be accessed from any class in the same package as `A`, as well as from any class derived from `A` (or derived from any class derived from `A`, *etc.*, regardless of package). However `m` cannot be accessed from any other class, while `p` can be accessed from any class.

Question 4 **[7 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 4

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 5 **[7 marks]**

Why should all instance variables be declared `private`? What could go wrong if this advice is not heeded?

Sample Answer to Question 5

Instance variables should be declared private so that no class other than the class they are defined in can access and modify them. If they are visible to other classes (as they would be with any visibility other than `private`), then the class cannot control access and modification.

This has two main problems. First, the class cannot control the sanity of its instances. For example, a date class with year, month and day instance variables would want to ensure that no invalid date could be created. If these instance variables were not private, the class could not ensure that no other class could modify a date to be February 31. If the instance variables are made private, then any methods to set the year, month, and day can ensure that dates are always valid.

The second problem with making instance variables public is that it is a failure of information hiding. All the public members of a class define the class's interface, and a class's interface should be fairly stable. The private members of a class can be removed or modified at will without affecting other classes, but changes to public members may require significant modification to other classes. Thus by making instance variables public, the class's author makes it difficult for him or her to later remove or change them. For example, if a `Date` class has public year, month and day instance variables, then changing the representation of dates would require every class that accesses the year, month and day to be modified. If year, month and day were instead accessed through methods, then these methods could be modified to suit the new representation, and no other classes would need to be changed.

Question 6 **[8 marks]**

What is *polymorphism* and why is it desirable? How can it be used in Java? Give some Java code fragments to illustrate.

Sample Answer to Question 6

Polymorphism, meaning *many shapes*, is the programming language feature that allows a single piece of code to work for many different types of objects. Java supports polymorphism in three different ways. First, *inheritance, overriding*, and *late binding* support polymorphism. Inheritance allows a class to be derived from a base class, adding extra instance variables and methods. The derived class may also replace, or override, methods of the base class with its own implementations of some of the base class's methods. In other methods, a variable whose type is the base class may hold an instance of either the base class or the derived class, and messages sent to this object will behave as appropriate for its actual class. This is called late binding. Thus a method may interact with an instance of either the base or derived class without needing to worry about which it is. For example, a Java `Employee` class may have `HourlyEmployee` and `SalariedEmployee` subclasses. Each subclass has its own method for calculating the pay appropriate for that class of employee. This allows a method to compute the pay for either an `HourlyEmployee` or a `SalariedEmployee` without needing to know which it is:

```
Employee e = ... // could be HourlyEmployee or SalariedEmployee
double pay = e.getPay(); // computes correct pay regardless
```

A second form of polymorphism in Java is *interface inheritance*, accomplished with Java interfaces. An *interface* is a Java type that is fully abstract; no instances of it can be created. However, any classes can declare that they `implement` that interface, which means that an instance of any class that implements the interface can be stored in a variable of that interface type. Then the methods of the interface can be used on that variable. For example, any type may be declared to implement the `Comparable` interface, which means that two objects of that type may be compared to determine which is smaller or larger than the other. Then an array of objects whose class implements `Comparable` can be sorted using a method that does not need to know the type of the elements, only that they implement `Comparable`. For example, an array of playing card ranks can be sorted using the static `Arrays.sort` method, which works for an array of any `Comparable` class:

```
Rank[] ranks = ...;
Arrays.sort(ranks); // sort method doesn't care about type
                    // of array elements
```

Generics support a third kind of polymorphism called *parametric polymorphism.* This allows a class to require 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. The methods of the `ArrayList` class do not need to know or care about the type of its elements; they work for `ArrayList`s of any type. However, the Java compiler will ensure that the types of the elements are appropriate. For example:

```
ArrayList<String> a = new ArrayList<String>();
a.add("this is a string"); // this is fine
int n = a.get(0); // compile-time error
```

Question 7 **[6 marks]**

What is the distinction between *single inheritance* and *multiple inheritance*? Which form of inheritance does Java support?

Sample Answer to Question 7

Single inheritance means that each class can be derived from at most one base class, while multiple inheritance allows a class to be derived from more than one class. Java supports only single inheritance, since in Java, a class can only specify one class after the `extends` keyword.

Java does allow a limited form of multiple inheritance, because a class may declare that it `implements` multiple interfaces. However, this cannot be used to inherit any method implementations, only method interfaces.

Question 8 **[6 marks]**

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

Sample Answer to Question 8

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 9 **[8 marks]**

Write a method that takes an array of `int` as its only input and returns the average of the values in the array as a `double`.

Sample Answer to Question 9

```
public static double average(int[] data) {
    int total = 0;
    for (int i = 0; i < data.length; ++i) {
        total += data[i];
    }
    return (double)total / data.length;
}
```

— End of Paper —