Examiners' report 2009

2910212 Programming: advanced topics and techniques – Zone A

General remarks

Section A: JAVA

Part A of the examination consisted of five questions on object-oriented programming in Java, and candidates were asked to answer three questions. There were 25 marks available for each question, 75 marks in total from this section, with another 75 marks available for Section B.

In any examination, candidates may be asked to provide definitions of core concepts of the course (Classes and Objects; Methods; Data Abstraction; The Graphics2D API; Composition; Inheritance; Encapsulation; Design Patterns; GUIs) which can be found in the subject guide and recommended textbook. Candidates need good knowledge and understanding of these concepts and, in addition, with about half the marks awarded for writing code (64 out of 125 on this paper) it is also very important that candidates do the programming exercises in the subject guide in order to prepare for their examination.

Comments on specific questions

Question 1

Exceptions

Parts (a) and (b)

Most candidates recognised that *throws* is the key word used when declaring exceptions. However when asked, in part (b), to explain how exceptions are used for handling errors using the *absValue()* method as an example, many candidates just described specifically what the *try* and *catch* blocks in the method did in that instance. For full credit, a general description of how *try* and *catch* are used to handle errors, with *absValue()* used as an example was needed.

Candidates could have written something like: Errors are handled with use of the keywords *try* and *catch*. *Try* and *catch* must be used with *throw,* so that errors will be thrown, as they must be before they can be caught. Firstly the code to be executed if there is no error is enclosed in a *try* block. This is followed by a block of code allowing the program to recover from incorrect input identified with the keyword *catch*. If an exception is thrown the *catch* block of code will be executed, otherwise the *try* block will be. In the *absValue()* method firstly the *onlyPositive()* method is tried, if this throws an exception it is caught in the *catch* block. An exception will be thrown if a negative number has been entered and will be handled by multiplying the negative number by -1

to make it a positive number, allowing the program to recover from the error and continue.

Part (c)

Candidates were first asked to write a main method that would call the *absValue()* method with two different values, storing the values returned in integer variables and then printing them onto the screen. This is quite basic code writing and while many candidates found it easy and wrote completely correct answers, some created an instance of the class in order to call the method – while this is not wrong these candidates were failing to recognise that the method was static so it could have been called without instantiating the class. In addition, quite a number of candidates had no idea how to write a main method, something that should be second nature by now. The code needed was very simple:

```
public static void main (String [] args)
{
        int x = absValue(-3);
        int y = absValue(3);
        System.out.println(x);
        System.out.println(y);
}
```

Candidates were then asked what their program would output to the screen. A large number of candidates thought that the answer was:

3

-3

This shows an inability to read code – the *absValue()* method first tries to use *onlyPositive()*, if an error is thrown because a negative number has been entered *absValue()* catches the error and handles it by returning the number multiplied by -1, thus making it positive. Hence

3

3

would be printed out by the above main method.

Candidates were then asked what would happen if you compiled your program with the statement *absValue(3.0);* in your main method. Most correctly said that the program would not compile because of a type mismatch (ie *absValue()* expects an int not a double). However, a large minority lost marks because they thought that this would cause a runtime error. Some just said this would cause an error, but for full credit candidates needed to correctly specify the type of error – compiler error.

Part (d)

Some candidates lost marks on this question because while they identified the problem correctly (no base case) they did not explain how the method would behave with no base case, as they should have done for full marks.

The question was looking for candidates to show their understanding that the *power()* method, since it was recursive, would keep calling itself but with the parameter *b* reduced by one each time. Eventually *power()* would be called with -1, and this would trigger throwing the *NegativeException*. This meant that whatever number was entered, the method would always throw the exception. To correct this, a base case is needed, to terminate the recursion, ie *if (b==0) return 1;*

Question 2

Design patterns

Part (a)

Candidates were asked to explain how design patterns help in the reuse of ideas. While some good answers were seen, for full marks candidates needed to be clear that design patterns are not a Java concept, but are independent of programming languages. A good answer would have said that design patterns are ways of encapsulating useful ideas behind program design, allowing for the reuse of ideas, concepts, and solution plans. Design patterns primarily help us to understand good solutions to recurring programming problems. Object-oriented design patterns usually describe relationships between classes and objects and can be implemented in any object-oriented language.

Part (b)

This question was answered very badly with far too many candidates clearly describing composition (for no credit) and not the composite design pattern. A good answer could have said: The idea of the Composite pattern is that it allows us to manipulate composites in exactly the same way we manipulate primitive objects. For example, graphic primitives such as lines or text must be drawn, moved, and resized. But we also want to perform the same operation on composites, such as drawings, that are composed of those primitives. Ideally, we'd like to perform operations on both primitive objects and composites in exactly the same manner, without distinguishing between the two. This is the problem that the composite design pattern helps us to solve.

Part (c)

Most candidates answered this question by writing code, where some credit was given for showing knowledge of the *Iterator* interface and its methods. Others answered the question by defining the iterator design pattern which is not what was asked for and no credit was given. All that was needed for full marks was to say that Java implements the iterator design pattern with an *Iterator* interface that has three methods: *next()*, *hasNext()* and *remove()*.

Part (c)(i) and (ii)

Most candidates correctly named the template pattern as being the one that the *Induction* class was implementing, but only a very small number of good definitions were seen. A good answer would have said something like: The template design pattern defines a computation as a fixed sequence of steps, with one or more of the steps variable; and is implemented in Java by having an abstract class with most methods

implemented, except for one or two abstract methods that represent the piece that has to be filled in when using the template.

Part (ii) was answered correctly by the majority, but some strange answers were seen where candidates clearly did not know the syntax of a method or a class. These candidates would have improved their performance in the examination if they had spent time doing the exercises in the subject guide and becoming familiar with standard Java code.

Question 3

Inheritance

Part (a)

Most candidates got the majority of the marks for this question, but lost one or two marks by not making clear that with inheritance for extension, new methods are added without overriding any of the existing methods (a key distinction between it and inheritance for specialisation); or by talking about overridding in the context of inheritance for specialisation without making it clear why methods are overridden.

A good answer might have said: In inheritance for extension a subclass extends an existing class in order to define new behaviours to supplement those it inherits. New methods are defined without overriding any of the existing methods, and the subclass inherits both interface and implementation; In inheritance for specialisation a subclass extends an existing class in order to modify some of the class's behaviour by overriding some of its methods. Hence behaviour is partly overridden and partly inherited, however the interface is inherited; In inheritance for specification some or all of the inherited methods are abstract, inherited ideally from an interface, or from an abstract class. With an interface no implementation is inherited, with an abstract class some may be.

Part (b)(i)

This was done well, but the most common error seen was this:

```
public class duckBilledPlatypus extends Mammal
{
        String reproduce()
        {
        System.out.println ("This mammal lays eggs.");
        // above should be return not s.o.p
        // eg return ("This mammal lays eggs.");
        }
}
```

The *reproduce()* method is of type String, and so it should return a string.

Part (b)(ii)

Most candidates correctly answered *specification* but needed to give a complete justification (and a clear one) for full marks, for example: The *duckBilledPlatypus* subclass demonstrates inheritance for specification as it implements behaviour that has been left abstract by the parent class.

Part (b)(iii)

The question somewhat misleadingly asked candidates to implement the *reproduce()* method by printing out the string "This mammal gives birth to live young." when in fact the method, which was of type String, should return a string, not print one out. Full credit was given to candidates who either returned or printed out a string, and also to those candidates who did both – as in the following possible answer:

```
public class human extends Mammal
{
        boolean hasFur()
        {
                return false;
        }

        String reproduce()
        {
String s = "This mammal gives birth to live young."
System.out.println(s);
return s;
        }
}
```

However, a few poor answers were seen where candidates had no grasp of Java syntax. For instance, they did not know how to implement an abstract method, and put the class names the wrong way round in the declaration ie *class Mammal extends human*. These candidates had clearly not practised writing programs.

Part (b)(iv)

Most candidates correctly answered that the *human* subclass demonstrated both inheritance for specialisation and for specification but some lost marks by not clearly and completely justifying their answer as the question asked. Usually these candidates provided some justification but it was unclear whether their justification applied to specialisation or specification. A clear and simple answer would be: The subclass *human* shows inheritance for specialisation since while most of the existing behaviour of the superclass *Mammal* is kept one method, *hasFur(),* is overridden. However, the subclass also demonstrates inheritance for specialisation as it implements behaviour that has been left abstract by the parent class.

Question 4

Methods

Part (a)

Many candidates found this question fairly easy and gained full marks. However, some lost marks by using a loop in the *swap()* method. I would assume that these candidates are thinking that every method that takes an array has to iterate over the entire array. In fact, by carefully reading the question (as most did) candidates would learn that the method only has to operate on two items in the array, hence no loop is needed. Some candidates made the mistake of swapping *i* and *j* rather than the array elements denoted by *i* and *j*. The correct code is as follows:

```
public static void swap(int[]a,int i, int j)
{
 int tmp = a[i];
 a[i] = a[j];
 a[j] = tmp;
}
```

Part (b)

Most candidates gained full marks for this, but a few made the mistake of thinking that *a* was modified by the *fact()* method. Some lost marks by including the *Modifies* comment as asked, but not explaining why it was empty, hence the Examiner did not know if it was empty because the candidates understood that nothing was modified or was it empty because the candidate did not know how to answer? Others added an explanation that the *Modifies* clause was not needed as nothing was modified, or wrote 'null' or 'nothing modified' in the comment, and gained full marks.

Additionally, some candidates wrote complicated comments, particularly for *Effects,* when all that was needed was something very simple demonstrating understanding of the uses of the three comments, for example:

```
public static int fact (int a)
{
        // REQUIRES: 0 <= a
        // EFFECTS : Returns the factorial of a
        // MODIFIES: nothing
}
```

Part (c)

This was done well on the whole, but a small number of candidates need to note that where the question said no credit for iterative methods, it meant it! Additionally, a few candidates lost marks for not including a base case, or for mistakes in the recursive call. Once possible *fact()* method with correct code is below:

```
public static int fact(int a)
{
        if(a == 0) return 1; // base case
        else return (a * fact(a-1)); // recursive call
}
```

Question 5

GUIs

Candidates were required to write a Java program that displayed a rectangle in a 400 x 400 *JFrame*, and then to add a *JButton* that, when pressed, made the rectangle change size. This was quite a hard question and all credit goes to those who wrote very good programs under examination conditions. For this question, absolute syntactical correctness was not important, but demonstrating a good understanding of the different elements needed, ie the *ActionListener* (or *MouseListener*), constructing the object, the paint method, *JButtons* and *JFrames* etc, was. All credit was given to sensible ways of changing the rectangle, even if minor syntactical errors were seen.

A point for candidates to note is that if candidates put comments in their code it helps the Examiner to understand their intentions, and thus to give credit for good logic even if the code is syntactically incorrect and its purpose is unclear.

Section B

Question 6

This question concerns two special features of functions in SML, namely *infix* and *currying*.

Candidates should have had no difficulty in producing an inbuilt infix function, such as the many arithmetic ones, but some did not do this and so lost marks. It is good examination preparation to think of examples (both inbuilt and of your own making) of each of the many concepts introduced in the half unit. Examination answers are very often improved by the inclusion of appropriate examples – even those questions that do not explicitly ask for them.

Both *association* and *precedence* need to be considered when defining infix functions. These are less of an issue with the normal prefix functions, as brackets are almost always used in the latter and evaluation goes from right to left. However, they are both important when infix functions are in use, as there are ambiguities unless brackets are included.

As an example of this, the function *p* which given two integers (*x* and *y*, say), returns the difference between their cubes, is used to allow candidates to describe the syntax required to convert a function into an *infix* one. Explanation of how operator precedence issues are dealt with is required for full marks.

Two different results are expected of the evaluation of ***1 p 2 p 3*** because:

$(1^3 - 2^3)^3 - 3^3$ is different from $1^3 - (2^3 - 3^3)^3$.

Unfortunately, most candidates failed to evaluate the superscript outside the brackets and so obtained incorrect results.

The concept of currying a function is quite a straightforward one but is not often met in traditional programming languages. The omission of brackets surrounding the function's parameter list is the main evidence of currying. This is in line with the strict typing that SML uses most of the time.

Given two functions, one that can be curried and one that cannot, candidates were asked to give the results of the evaluation of simple expressions including them. This was reasonably well answered but errors that were made, indicated that candidates were not familiar with evaluating partially formed expressions. It would be wise if candidates spent some time considering the practical differences between these two types of function definition before the examination. The use of SML to evaluate such expressions is a good learning tool.

Question 7

The feature of strong typing SML is explored in this question. Candidates should know examples of where this impacts on the language, as well as examples of where SML does not live up to its ideal.

In part b) the SML code for a user-defined data type is given. Candidates are assumed not to have met the data type before, but they must understand the definitions used for user-defined data types.

Candidates are invited to explain the meaning and use of each line of the SML code. It is not sufficient to translate the symbols directly into English. What is required is an indication that the candidate understands how the line works.

Having explained the definition, candidates are then required, in part ii) to give the output produced by a number of evaluations. Although some answers were correct here, a large number of candidates were unaware that the data type name resulted from many evaluations. This shows a lack of practical experience in the use of SML as a means to understand the concepts of the half-unit.

The final part of the question required a few lines of SML to be written. Candidates who understood the given code should not have had much difficulty in obtaining most of the marks for this part. However, those who were not familiar with user-defined functions struggled to obtain many marks.

Question 8

This question allows candidates to illustrate their mastery of the concepts of *List*, *Tuple* and *Record*, giving examples of their definition and use. It was disappointing that many candidates were only able to write about Lists and had very sketchy knowledge of Tuples and Records.

The rest of the question expected candidates to write simple code using the Record type. Unfortunately, many did not spend enough time thinking about their strategy and thus ended up with only sketches of code.

Question 9

Like SML, Prolog relies on pattern matching and Question 9 asks for the rules for matching terms. Many were able to write down the three or so rules required, but some answers rambled on without giving anything resembling a rule.

Once these rules are mastered, it should be easy to use them to see whether two expressions match. This part was done well on the whole, but many candidates would have benefitted from a pre-examination exercise in the application of the rules.

The Prolog predicate given to study in part b) proved too hard for many candidates. Some practice in taking a definition of a predicate, understanding the action of each line and then the action of the whole, is well worth the effort before the examination.

The final part of this question requires a predicate that *cycles* a list. Candidates who heeded the hint given used *append* to write the required predicate in a few lines of Prolog.

Question 10

The final question asks candidates to discuss the implications of the lack of strong typing in Prolog. These implications are given in the question quoted from the guide.

Candidates are required to justify the advantages and state some of the disadvantages. Good candidates were able to compare their experiences in SML with their experiences of Prolog and give a good account of the issues involved.

The declarative and the procedural readings of a Prolog program are needed when developing any non-trivial Prolog program. Candidates should be aware of this need and of the differences between the two readings.