# Examiners' commentary 2018–2019

## CO1109 Introduction to Java and Object-Oriented Programming – Zone B

### General remarks

The examination was attempted well by the majority of candidates, with some candidates showing an excellent grasp of Java that went beyond the basics. A minority of candidates would have been well advised to do some basic preparation for the examination including reading the subject guide (both volumes) and attempting the exercises within it. Programming cannot be learned without practice.

### Comments on specific questions

#### Question 1

This was a popular question, attempted by almost all candidates.

**a. Answer**

i. (A) `while` loop

(B) `for` loop

(C) no loop, single statement needed

(D) nested `for` loops

ii. `for(int i=1;i<=x;i++) System.out.println(i);`

**OR**

`for(int i=1;i<x+1;i++) System.out.println(i);`

**OR**

`for(int i=0;i<x;i++) System.out.println(i+1);`

iii. (A) Infinite loop – the program will hang with no output

**Comments**

In (i) most candidates answered correctly, with a few common errors seen.

(A) Some candidates answered 'no loop', but the key to answering the question is the word 'valid' as in 'asking the user to enter a *valid* file name': what happens if the user enters the name of a file that cannot be found? A `while` loop is the best choice of the ones available, as, if properly implemented, it should mean that the loop will continue only until a valid entry is made.

(B) Some answered `while` loop, which is possible, but a `for` loop is most appropriate for iterating a fixed number of times, and a loop to display the contents of an `Array` would iterate as many times as the `Array` has entries, determined by the *length* variable from the `Array` class.

(C) Some candidates thought that a `while` or `for` loop would be most appropriate here, seeming not to notice that printing the first entry in an `Array` could be done in a single statement, with no loop needed, for example, if the `Array` was called 'a': `System.out.println(a[0]);`

(D) Most candidates understood that nested `for` loops would be the most appropriate solution here.

In part (ii) the majority answered correctly, with one of the statements above.

In (iii) a few candidates thought that there would be no output since the program would not compile, and many candidates answered (D) *None of the above*. These candidates need to note that `while(true)` is a valid `while` loop – see section 8.7.3 of Volume 1 of the subject guide.

**b. Answer**

i. 10 stars
ii. 8 stars
iii. 8 stars
iv. no output
v. INFINITE
vi. 6 stars
vii. 3 stars
viii. INFINITE
ix. INFINITE

**Comments**

Part (b) was often completely right, and all candidates gained most of the marks for this question. Common mistakes were off-by-one errors and thinking that the loop given in (iv) was infinite, and that the loop given in (ix) had no output, or would output 1 asterisk.

**c. Answer**

```
private static void mainLoop() {
   while (keepPlaying) {//should use keepPlaying
      showMenu();
      askUserToChoose();
      int choice = getUserChoice();
      executeChoice(choice);
   }
}//a do/while loop is also a valid answer
```

**Comments**

Some candidates gave the above answer, other correct answers collapsed the last two statements into one: `executeChoice(getUserChoice());`

Candidates were expected to notice that the *executeChoice()* method will make the *keepPlaying* variable `false`, once the method is given the number 3 as its `int` parameter, that is once the user has decided to quit. This means that once the user enters 3, the main loop will end, provided that it is continuing as long as *keepPlaying* is `true`. Some candidates used a local `boolean` variable to control the loop in the *mainLoop()* method. These candidates clearly understood that using a local `boolean` meant that the *mainLoop()* method had to end the loop appropriately, once the user decided to quit, and that this could be done by changing the state of the `boolean`. Unfortunately, this was often implemented badly, with candidates making the variable `false` once the user had entered any valid number. This would mean that the loop would always end after one user choice had been executed, clearly a logical error.

Other common errors were:

- writing a method with the four statements given above, but not enclosed in any loop at all.
- writing the guard for the `while` loop as `(keepPlaying = true)` – one of the most common Java errors. A single equals sign is used for

assignment, so the above statement is making the *keepPlaying* variable `true`. Two equals signs test for equality, as in `(keepPlaying == true)`. Simpler still is just to use `while(keepPlaying)` which the compiler understands to mean while the `boolean` variable in the brackets is true.

- not understanding that the ouput of the *getUserChoice()* method needed to be given to the *executeChoice()* method, hence writing the following two statements: **getUserChoice(); executeChoice();**
- not capturing the output of the *getUserChoice()* method in a variable, but still giving the *executeChoice()* method a parameter, *i.e.* the statements were: **getUserChoice(); executeChoice(choice);**

A small number of candidates handled the potential `NumberFormatException` that could be thrown by the *getUserChoice()* method if the user's entry could not be parsed to an `int`. This was not necessary for full credit, but was an excellent idea.

## Question 2

This was a popular question, attempted by nearly all candidates.

**a. Answer**

i. (B) Correct the first error only and recompile

ii. (A) `10 10`

iii. (B) `non-static variable z cannot be referenced from a static context`

**Comments**

Most candidates thought that the correct answer to part (i) was *(C) correct all errors and recompile*. The subject guide, Volume 1, section 2.8.1, states:

The best way to correct [compilation errors] is just to correct the first one and then recompile. This is because the first error sometimes makes the compiler think there are lots of other errors which are not really there.

Most answers to (ii) were incorrect, with the most popular answer being C (`5 10`), closely followed by B (`10 5`). This demonstrated that most candidates did not understand that the *swap()* method in the *Q2* class was logically incorrect, and would not swap the values contained in the variables *x* and *y*, but would instead make the two variables equal to each other. Firstly, the method first makes *x* equal to *y*, so *x* and *y* now have the value 10. Then the method makes *y* equal to *x*, so *y* gets the value contained in *x*, which is 10. Since *y* is already 10 this makes no difference to the output. In order to successfully swap the values contained by the two variables, firstly the variable contained in *x* would need to be saved, then *x* made equal to *y*, then *y* should be made equal to the saved value from *x*. Hence a correct *swap()* method in the *Q2* class would look like this:

```
static void swap (){
   int temp = x;    //save the value of x (which is 5)
   x = y;           //x equals 10
   y = temp;        //y now equals 5
}
```

Part (iii) was usually answered correctly, although a large minority thought that the answer was either C (*No error – the class will compile*), D (*None of the above*) or A (`error: incompatible types: unexpected return value`). Since incorrect answers were spread fairly evenly across the three alternatives, it is likely that most of these candidates were guessing.

**b. Answer**

i. NO

ii. NO

iii. YES

iv. YES

v. YES

vi. YES

vii. YES

viii. YES

ix. YES

**Comments**

In part (b) most candidates answered entirely correctly, with every candidate achieving at least 5 marks. There was one common error, a minority thought that `Integer.parseInt(15);` would type check, which it will not as the *parseInt()* method expects a `String` parameter, and a literal `String` must be enclosed in double quotes.

**c. Answer**

- *two(String)* – ***reverseString*** or similar (1 mark)
  (prints the `String` parameter with the characters in reverse order)
- *three(String)* – ***randomiseString*** or similar (2 marks)(prints the `String` parameter with the characters in random order)
- *four(String*) – (***printVowelsOnlyInUpperCase*** or similar) (1 mark)
  (prints the `String` in upper case with all consonants removed/prints only the vowels of the `String` in upper case)
- *five(String)* – ***printUpperCaseWithVowelsRemoved*** or similar (1 mark)
  (prints the `String` parameter with all vowels removed in upper case/ prints only the consonants of the `String` parameter in upper case)
- *six(String)* – **addASpaceAfterEachCharExceptLastOne** or similar (1 mark for *adding a space*, 2 marks *for except for the final char*)
  (prints the `String` parameter with a space after each character except for the final character)

**Comments**

Note that the answers given are model answers, equivalent answers were seen and given full credit. For example, names proposed for *three()* included *scrambleString*, and *shuffleLettersInString*; answers such as these were given full credit.

Note that the comments with the above model answers are not part of the answer expected from candidates, only the method names given in bold are included in the model answer.

The most common errors were:

- In the examination paper, the method *three()* had a comment 'swap characters'. Most candidates gave a new name to *three()* that referenced this comment. Names such as *swapString()* or *swapChars()* gained no credit. Some candidates gave a new name to *three()* that suggested that the candidate thought that swapping two random chars from the `String` parameter happened only once, for example *showWordWithRandomLetterSwap()*. These candidates did not recognise, or did not show in their answer that they recognised, that the swapping was in fact being done in a `for` loop that iterated as many times as the `String` had `chars`, such that the output was the letters from the parameter `String` in random order.

- In the *three()* method, the parameter `String` was first put into an `Array` of `chars`, and then elements in the `Array` were swapped by randomly generating two numbers and then swapping the places of the elements at those index numbers. This was done for as many times as there were letters in the parameter `String`, then the contents of the `Array` were printed. Some candidates referenced the use of an `Array` in their answers, for example *arraySwap()* or *swapCharsInArray()*. Such answers focussed on the implementation, when they should have focussed on the output.
- Most candidates recognised that *four()* was printing only the vowels of the parameter `String`, but did not recognise that the vowels would be output in upper case.
- Most candidates recognised that *five()* was removing the vowels of the parameter `String` and outputting only the consonants, but did not recognise that the output would be in upper case.
- Some candidates failed to show in their answers that they understood that a space was only inserted between letters in the `String`. Answers that left open the possibility that a space might be added after the final character, did not receive full credit.

## Question 3

This was a popular question, attempted by most candidates.

**a. Answer**

i. (B) The class will compile and output `true`

ii. (A) `true`

iii.
```
if (x <10 || y < 10) System.out.println("true");
   else System.out.println("false");
```

**Comments**

Most candidates thought that the class *Bool* from part (i) would not compile, and/or that the class *Bool5* from part (ii) would not compile, while a few chose D (*None of the above*) for (ii). These candidates clearly needed more practice with `boolean` variables and Boolean expressions. I would suggest reading section 7.11 of Volume 1 of the subject guide, and attempting the exercises in section 7.12.

Some odd answers were seen to (iii), including putting the original statements into a `while` loop. Some candidates must have misread the three statements given on the examination paper, writing answers such as:

```
if (x <10 || !(y < 10)) System.out.println("true");
else System.out.println("false");
```

which of course means if *x* is less than 10, or *y* is not less than 10 print `true`, in all other cases print `false`. So when *x* is less than 10, or *y* is greater than or equal to 10 `true` is printed, which is correct. However, if *x* is greater than or equal to 10 but *y* is less than, `false` will be printed, which is incorrect.

Some candidates used AND (`&&`) instead of OR (`||`), for example:

```
if (x < 10 && y < 10) System.out.println("true");
else System.out.println("false");
```

The above if/else is incorrect, as `true` will only be printed when both variables are less than 10, but `true` should be printed if either one of them is, irrespective of the value of the other.

Consider the original expression:

```
if (x < 10) System.out.println("true");
else if (y<10) System.out.println("true");
else System.out.println("false");
```

Let us consider the possible output from the above three statements.

4. *x* less than 10 (*y* greater than or equal to 10) : TRUE
5. *x* less than 10 (*y* less than 10) : TRUE
6. *x* greater than or equal to 10, *y* less than 10 : TRUE
7. *x* greater than or equal to 10, *y* greater than or equal to 10: FALSE

From the above, clearly answers such as:

```
if (x < 10 && y < 10) System.out.println("true");
else System.out.println("false");
```

would not be correct, as `true` could only be the output when condition 2 in the above list was met, but not when conditions 1 and 3 were met. It was possible to give a correct statement using `&&` (AND), seen very rarely, as follows:

```
if (x >=10 && y >= 10) System.out.println("false");
else System.out.println("true");
```

The above statement will output `false` when both *x* and *y* are greater than or equal to 10, and will output `true` in all other cases, as it should.

One very good answer seen was:

```
boolean status = (x <10 || y < 10);
System.out.println(status);
```

**b. Answer**

| **i.** | **ii.** | **iii.** |
|---|---|---|
| What happens when the pressure is 30 is undefined. | `12` | /***1***/ and /***2***/ |

**Comments**

Just about half of the answers seen to part (i) were correct. Incorrect answers mostly focussed on how the use of so many `if` statements without `else` was not possible, although some candidates simply thought that the use of many `if` statements was inefficient. Perhaps, but inefficiency is not a logical error, neither is `if` without `else`. If it were not possible to use `if` without `else` (it is possible) that would be a syntactical error, rather than a logical one.

It is possible to view errors in our programs' code as being of 3 types:

- **Syntax** errors – errors that the compiler will pick up on.
- **Run-time** errors – those that happen only when the program is run.
- **Logical** errors – the program is syntactically correct, it runs without errors, but it does not do what it is supposed to do, because of some logical flaw in a method or algorithm.

Clearly forgetting to include an action for when the pressure variable *z* is equal to 30 is a logical error, by a process of elimination if nothing else: (1) the compiler will not pick it up as a syntax error; and (2) when the method is run, if control passes from `if` statement to `if` statement without any of them being executed, this is no reason to trigger a run-time error.

In addition to the most common wrong answer of claiming that `if` without `else` is inefficient, a handful of students showed a lack of basic understanding. Some wrote that Boolean conditional statements such as `(z`

`<= 100 && z >= 60)` were impossible, and others that the error was that the *z* variable had not been assigned a value. Such basic errors show a lack of serious preparation for the examination.

Part (ii) was usually correct, but a large minority answered with the wrong number, usually 9, but also 0, 11, 13, 14, 18 and 103. Consider the statement

```
int x = 2/3;
```

Since *x* is an `int` variable, the JVM will perform integer arithmetic, meaning that 2 divided by 3 is zero with 2 remainder. The remainder is thrown away, so the result of the division is zero. Hence x = 0 and y = 3, so the statement

```
z = 2*x +3*y;
```

will assign the value 2 x 0 + 3 x 3 = 9 to *z*. Thus, the result of *z*+*y* will be 9+3=12.

Part (iii) was always answered correctly.

**c. Answer**

```
*********
**       **
* *     * *
*   * *   *
*    *    *
*   * *   *
* *     * *
**       **
*********
```

**Comments**

Many correct answers were seen, but also quite a few incorrect answers. The most common incorrect answers were those that missed one or other of the diagonal lines. This was quite a challenging question, so those who answered correctly are congratulated.

## Question 4

This question was attempted by about 40 per cent of candidates.

**a. Answer**

i. (A) Because the *read()* method returns –1 when it detects the end of the file.

ii. (B) The program will output the text contained in the text file 'filey.java'.

iii. (B) The program will output the Unicode value of each character in the file.

iv. (B) There would be no output as the condition t!=-1 would be false at the beginning.

**Comments**

A few incorrect answers were seen to parts (i), (ii) and (iv), in each case either (C) or (D).

Most candidates thought that the answer to (iii) was (C), that the program would output the text in the file. This demonstrated that many candidates did not understand that the *filey* class was reading from the input file one `char` at a time, that each `char` was read as an `int`, and that the `int` value read would be the Unicode value assigned to the `char` read at that point in the file, and this `int` Unicode value was being cast to a `char`. Removing `(char)`

from the statement `System.out.print((char)t);` would mean that the Unicode value was not being cast to a `char`, and hence an `int` would be printed.

**a. Answer**

i. `closing program`

ii. `FileNotFoundException`

iii. `NumberFormatException`

**Comments**

Answers given to part (i) by a minority showed a lack of basic programming knowledge and experience, with some candidates writing that there would be no output. Candidates wrote that there would be no output because the file was empty, or because the program would not compile, or because an exception would be thrown. Of course, an exception would be thrown because the file did not exist, hence control would pass to the catch block, which would output `closing program.`

Most candidates gave the correct answer to part (ii), but struggled to come up with the correct name of the exception for part (iii), writing such things as `InputMismatchException,` or *ObjectMismatchException* or *TypeMismatchException*. Candidates who wrote such things as *NumberFormatMismatchException* and *NumberMismatchInputException* gained some credit.

**c. Answer**

```
public static void readFromFileAndAdd() {
      int fileInt;
      try {Scanner in = new Scanner(new
      FileReader("file1.txt"));
         while(in.hasNextLine()) {
            String line = in.nextLine();
            fileInt = Integer.parseInt(line);
            sum = sum + fileInt;
         }
         in.close();
      } catch (Exception e) {
            System.out.println("Error: unable to continue");
         }
}
//bold text was given with the question
```

**Comments**

Part (c) was attempted very badly by most candidates.

Note that candidates were expected to read and understand the *Q4C* class well enough to know that the *sum* variable needed to hold the result of the addition, as *sum* was a class variable that was used by the *writeResultToNewFile()* method to save the result of the addition to the output file. Despite this, many candidates made a local int to save the result of the addition to. This local variable would not be available to the *writeResultToNewFile()* method.

By far the most common error seen with answers to part (c) was copying the streams to read from the file from part (a), meaning that the numbers being added were the Unicode values of the `chars` representing the numbers in

the file, which is clearly incorrect. In the above model answer a `Scanner` has been chained to a `FileReader`, allowing the use of the `Scanner` class's *nextLine()* method to parse the input read by the `FileReader` one line at a time. Some good answers were seen, some that used the *nextInt()* method of `Scanner` class rather than the *nextLine()* method. The *nextLine()* method reads a `String`, which then has to be parsed to an `int`, while the *nextInt()* method reads in `ints` directly. However the *nextInt()* method can be problematic because it leaves a hanging new line; despite this, candidates implementing their methods in this way received full credit.

Other errors seen were forgetting to close the input file and not writing the method with try/catch blocks for exception handling.

## Question 5

This question was attempted by a little over one third of the candidates.

**a. Answer**

i. `char`

`double`

ii. (A) FALSE

(B) TRUE

iii. `3`

iv. `11`

**Comments**

There is a simple rule to distinguish primitive from reference variables in Java – all primitives start with a lower case letter. The answers given to part (i) show that about a third of candidates attempting the question did not know this simple rule, since so many included `Boolean` and/or `String` as primitive variables. Section 4.4 of Volume 1 of the subject guide lists the 8 primitive variable types, while section 5.4 of Volume 2 of the guide discusses simple (primitive) and reference variables.

A few candidates thought that statement (A) in part (ii) was true, but in fact this is true for reference variables, but not for primitives.

Some candidates thought that the output in part (iii) was `6` or `9`, and many thought that the output in part (iv) was `1`. These candidates should note that the *IntParam* class given in part (iii) was adapted from exercise 5.7.5 in Volume 2 of the subject guide, while the *ArrayParam* class given in part (iv) was taken from exercise 5.7.1 in Volume 2. In part (iii) a copy of the value in the local `int` parameter *n* is sent to the method *p(),* hence invoking *p()* has no effect on the value held in *n*. In part (iv) the address of the `Array` *a* is what is sent to the *p()* method. The method then uses this address to change the value of what is pointed at by the `Array` variable *a*, a reference variable.

**b. Answer**

i. (B) This is an example of method overloading and will not prevent the program from compiling and running.

ii. (C) The *FloatToChar* class will compile and output a `char` (in this case `Y`); the *IntToBoolean* class will not compile.

iii. (A)
```
120
x
```

**Comments**

Part (i) was mostly answered correctly, with a small number of candidates answering A (*The program will have a run-time error because of a name clash with the two print methods*).

About 80 per cent of candidates answered part (ii) incorrectly, with (D) being the most popular answer, followed by (B), meaning that most candidates thought that the *FloatToChar* class would not compile, with more than half of those believing that the *IntToBoolean* class would compile. In fact an `int` cannot be cast to a `boolean`, since there are just 2 `boolean` values and billions of potential `int` values, how would it be possible to decide which of the two `boolean` values an `int` should have in a logically consistent way? Whereas a `float` can be cast to a `char`, so the *FloatToChar* class will compile. Since `chars` have Unicode values that are unsigned numbers, the JVM can discard the fractional part of the `float` (the part after the decimal point) and turn the `int` remainder into a `char`. This may not always work sensibly, as `int` covers a greater range than `char`, in particular, we cannot have negative Unicode values while we can have negative `int` values, however casting a `float` to a `char` will not cause a compilation error, as candidates who had attempted the exercises in Chapter 6 of Volume 2 of the subject guide would know.

About half of candidates answered part (iii) correctly. Candidates were expected to understand that (A) was correct, since the output would be an int on the first line, with a char on the next, and only (A) met these conditions. This is because the first statement in the *Chars* class adds up the Unicode values of the two chars then prints the resulting number, and the second statement also adds up the Unicode values of the two chars, but then casts the resulting number to a char. Naturally candidates were not expected to know what the particular int and char values output would be.

It is worth noting that the *Chars* class was adapted from the *AplusB* class in Volume 2 of the subject guide; see section 6.6 *Type Casting*.

**c. Answer**

```
private static void
   getPlainTextAndShiftFromUserEncryptAndShowResult(){
      askUserForTextToEncrypt();
      String text = getTextFromUser();
      askUserForShift();
      int shift = getNumberFromUser();
      String encrypted = CaeserCypher.encrypt(text,
      shift);
      showEncryptedResults(encrypted);
}
```

**Comments**

Part (c) was answered quite well, much better than part (b), with two common errors seen:

- Putting the statement given with the question: `String encrypted = CaeserCypher.encrypt(text, shift);` at the start of the method. When the method starts, the variables *text* and *shift* do not exist, so this would lead to compilation errors. This was a basic error and led to a complete loss of credit.
- Invoking the two methods, *getTextFromUser()* and *getNumberFromUser(),* without assigning the values returned by the methods to variables, so that the compiler would not be able to find the variables *text* and *shift* when it came to the `String encrypted = CaeserCypher.encrypt(text, shift);` statement.

## Question 6

This question was attempted by a just over half of candidates.

**a. Answer**

(A) A constructor must have the same name as the name of its containing class **TRUE**

(B) Constructors do not have return types **TRUE**

(C) A constructor is typically used to give values to the object's instance variables **TRUE**

(D) A class can have up to three constructors **FALSE**

(E) When one class extends another, the keyword `super` can be used in a constructor of the extending class to access a constructor in the class that is being extended **TRUE**

(F) An instance method has the keyword `static` in its heading **FALSE**

(G) `Person extends SentientBeing` means that the *Person* class can use the public instance methods of the *SentientBeing* class **TRUE**

(H) An inheriting class can redefine public instance methods from its parent class by overriding them **TRUE**

**Comments**

In part (a) quite a few candidates thought that (A) was false, while quite a few more thought that (C) was false, showing a lack of comprehension of the syntax and purpose of constructors. (F), (G) and (H) were also answered incorrectly by quite a few candidates. In general, many candidates demonstrated a lack of understanding of objects and inheritance in their answers to this question.

**b. Answer**

Most candidates gave a correct answer, either giving the fragment numbers in the correct order, or writing their answers out in full.

Some candidates gave the fragment numbers in the correct order as their answer, which, for full credit, could be:

4, 6, [2, 5], **[1, 3], [9, 12]**, **7,** [10, 8, 13, 11]

Methods are 1 & 3 (*toString*); 9 & 12 (*isSent*) and 7 (*printArray*); Their order is arbitrary, so they could appear in any order in the above.

Some candidates wrote out their answer in full, an example follows:

```
4.  public class Birthday {

6.    private String name;
      private String birthday;
      boolean cardSent;

2.    public Birthday(String name, String birthday, boolean
        cardSent) {

5.      this.name = name;
        this.birthday=birthday;
        this.cardSent = cardSent;
      }
```

```
1.      public String toString(){
          String s = ("The birthday of "+name+" is on ");
          s = s+(birthday+".");
          s = s+(" Card sent: "+isSent());
3.        return s;
        }
9.      private boolean isSent() {
12.       return cardSent;
        }
7.      public static void printArray(Birthday[] b){
          for(int i = 0; i<b.length; i++)
            if (b[i] != null)
              System.out.println(b[i]);
        }
10.     public static void main(String[] args) {
        //test statements
8.        Birthday adam=new Birthday("Adam","12 May 1960",
            true);
          Birthday eve=new Birthday("Eve","16 Oct 1958",
            false);
13.       Birthday[] birthdayList = new Birthday[10];
          birthdayList[0] = adam;
          birthdayList[1] = eve;
9.        printArray(birthdayList);
        }
      }
```

**Comments**

Most candidates answered this question correctly. Common errors were:

- Putting fragments 8 and 13, comprising the statements that make two new *Birthday* objects and adding them to an Array, into the constructor. Hence two statements that invoked the constructor were in the constructor. These statements should have been in the main method, where they are part of the test statements.
- In the main method, putting fragment 13 before fragment 11, which would mean that first the new objects were added to the Array, and then they were made. This would cause compilation errors; the compiler would say it was unable to find the symbols *adam* and *eve*.

Both of these errors are somewhat basic and suggest the candidates making them could have been much better prepared for the examination.

**c. Answer**

```
public class Question {

      private boolean answer;
      private String question;

      public Question(String question, boolean answer) {
         this.question = question;
         this.answer = answer;
      }

      public String getQuestion() {
         return question;
      }

      public boolean getAnswer() {
         return answer;
      }

}
```

**Comments**

A common error with answers to part (c) was not including the *getQuestion()* and *getAnswer()* methods. Candidates were expected to read the *quiz()* method, and to understand from the following two statements:

```
boolean answer=getAnswerFromUser(questions[i].getQuestion());
if (answer == questions[i].getAnswer()) correctAnswers++;
```

that the *Question* class needed a *getQuestion()* and a *getAnswer()* method.

For some reason a few candidates wrote their class with only a String instance variable, but in general this question was answered either very well or very badly, or was not attempted at all.