# Examiners' commentary 2018–2019

## CO1109 Introduction to Java and object-oriented programming – Zone A

### 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) no loop, single statement needed

(C) `for` loop

(D) nested `for` loops

ii. (C)
```
for(int i=x-1;i>=0;i--) System.out.
println(i);
```
OR
```
for(int i=x-1;i>-1;i--) System.out.
println(i);
```
OR
```
for(int i=x;i>0;i--) System.out.
println(i-1);
```

iii. (B) Infinite loop – the standard output will fill up with "Hello world", one to a line, and this will continue indefinitely.

**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 make a valid choice from a list of numbered menu items': what happens if the user makes an invalid choice? The user might type in too high a number, or accidentally enter something that cannot be parsed to an `int`. 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. Another answer sometimes seen was a for loop, which could be used but is not the best choice, given that the loop should be open-ended, continuing until a valid entry is given, then ending immediately once it has.

(B) Some answered `while` loop, seeming not to notice that printing the last 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[a.length-1]);`

(C) Most candidates understood a `for` loop would be the best solution, although some put `while`. However, adding together all the items in an `Array` could certainly be done with a `while` loop, but a `for` loop is most appropriate, as the loop would need to iterate through the first, 0-indexed item in the `Array`, to the last item given by the length of the `Array` minus one. In other words the loop would need to iterate as many times as the `Array` has entries, and this is most easily done with a `for` loop.

(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 expressions above. Some candidates did not seem to properly grasp the question, writing such things as `for(int i=x;i<=0;i--) System.out.println(i);`, which would output nothing if the user entered a positive number greater than zero, and would enter an infinite loop if the user entered zero or a negative number. In other words, some candidates simply changed the greater than sign (>) to less than (<), or less than or equals to (<=) which did not show much understanding of `for` loops.

In (iii) a few candidates thought that there would be no output since the program would not compile, but most answered correctly, understanding that `while(true)` was syntactically correct, and would give an infinite loop; that is, a loop that could potentially continue as long as the memory of the machine it was running on would permit it to.

**b. Answer**

i. 9 stars
ii. 6 stars
iii. 6 stars
iv. no output
v. INFINITE
vi. 5 stars
vii. 5 stars
viii. INFINITE
ix. INFINITE

**Comments**

Part (b) was often completely right, common mistakes were off-by-one errors and thinking that an infinite loop had no output, or that a loop with no output was infinite. Despite errors, all candidates received most or all of the credit for this question.

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

Most candidates understood that 4 methods needed to be called, and in what order to call them. Among these candidates there were two very common errors. The first, seen many times, was not enclosing the 4 statements in a loop. The second was not capturing the variable returned by the getUserChoice() method in an `int` variable, and not sending that `int` variable into the executeChoice() method. In other words, candidates wrote:

```
showMenu();
askUserToChoose();
getUserChoice();
executeChoice();
```

Other errors seen were not using the *keepPlaying* boolean variable to control the loop, and attempting to do things in the loop that were taken care of in the rest of the program, such as telling the user that their entry was invalid. Some candidates attempted to limit the entries that the user could make to 1, 2 or 3, which was a correct reading of the program, but bad programming practice. It was correct in that only these three numbers were valid menu choices, but incorrect in that the program dealt with incorrect entries in a way that allowed further menu choices to be added easily. Restricting the user's choice to only three possibilities would mean rewriting the method to allow for an extra choice if one were added. Otherwise another choice could be added to the *executeChoice()* method without needing to change anything else in the program.

## Question 2

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

**a. Answer**

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

ii. (B) `world world`

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

**Comments**

Part (a) was usually answered correctly, although a minority 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.

Mostly correct answers to (ii) showed that most candidates understood that the *swap()* method would not work to swap the values contained in the variables *x* and *y*, but would instead make them equal to each other, and the almost entirely correct answers seen to (iii) showed that candidates understood that instance variables must be accessed with instance methods, or with an object of the class and dot notation, and cannot be directly addressed by the main method.

**b. Answer**

i. NO `[Math.abs("Camelot");]`

ii. NO `[Integer.parseInt(10);]`

iii. YES `[int a = Math.abs("Camelot".length());]`

iv. YES `[Integer.parseInt("350");]`

v. YES `["threefifty".compareTo("350");]`

vi. YES `[int z = "threefifty".compareTo("350");]`

vii. YES `["boy".replace('b',"soup".charAt(0));]`

viii. YES `[Math.max(("hello" + " dog").length(),5);]`

ix. YES `[("boy".replace('b',"soup".charAt(0))).length();]`

**Comments**

In part (b) many candidates answered entirely correctly, with every candidate achieving at least 5 marks. There was one common error, a minority thought that `Integer.parseInt(10);` would type check. These candidates also tended to answer that (iv) would not type check, showing their confusion about what the *parseInt()* method does – it takes a `String` and returns an `int`. Hence the expression in (ii) will not type check as 10 is an `int`, and the method expects a `String`. In order to type check there would need to be double quotes around the 10, to show that it is a `String`, i.e. `Integer.parseInt("10");` would type check.

**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 *mixLettersString*; 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.

Most candidates understood that *two()* was printing its `String` parameter with the characters in reverse order, but few candidates understood that *three()* was randomising the order of the characters in its `String` parameter. Many gave the method a name such as *PrintWithRandomLetterSwap* which suggested that the candidate had failed to understand that the swapping process was in a loop, and happened more than once. In addition, the effect of the swapping of random characters in the `String` parameter, for as many times as the number of chars in the parameter word, had the effect of randomising the order of `chars` in the word, and candidates needed to show in their answers that they understood this.

Candidates usually understood that *four()* was printing only the vowels of a word, but often failed to appreciate that this would be in upper case (for example given the word "hello" the output would be "EO"). Similarly most candidates understood that *five()* was printing its `String` parameter with the vowels removed, but failed to understand that this would be done in upper

case. For example, given the word "hello" the output would be "HLL".

Many candidates lost marks for *six()* by not showing in their answers that they understood that the method only inserted a space between `chars`, and did not add a space after the final `char` of the word. In the method, the statements `c = word.charAt(length); s += c;` are used after the `for` loop in order to add the final `char` of the `String` parameter to the end of the new *String s* without a space after it. A significant minority of candidates thought that this was, in fact, adding the final character of the parameter word to the end of the new String *s,* twice.

## Question 3

This was a popular question, attempted by most candidates.

**a. Answer**

i. (C) The class will compile and output `false`

ii. (B) `false`

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

**Comments**

Most candidates gave the correct answers to parts (i) and (ii), although a few thought that the class *Bool* from part (i) would not compile, and/or that the class *Bool5* from part (ii) would not compile.

Candidates found part (iii) much more problematic, with some producing expressions that would not give the same result as the original. Quite often these candidates used AND (`&&`) instead of OR (`||`). 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* greater than 10 (*y* can have any value) : TRUE
5. *x* less than or equal to 10, *y* greater than 10 : TRUE
6. *x* greater than 10 and *y* greater than 10 : TRUE
7. *x* less than or equal to 10, *y* less 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 3 in the above list was met, but not when conditions 1 and 2 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");
```

**b. Answer**

| **i.** | **ii.** | **iii.** |
|---|---|---|
| What happens when the pressure is 60 is undefined. | `10` | /***1***/ and /***3***/ |

**Comments**

Quite a number of incorrect answers to part (i) were seen, with many candidates focussing on the `if` statements, and writing that the use of `if` without `else` was incorrect, or that there were too many `if` statements.

Some candidates gave no answer to this part of the question, while others focussed on how there was no output for negative values, even though the question explicitly said that pressure values could not be negative.

Part (ii) was often answered with the wrong number, 11 was a popular choice, but 7, 8, 9, 12 and 14 were also seen, while other candidates wrote that due to a type error, the *K2* class would not compile. In fact, the statement

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

would assign the value 1 to *x*, since the JVM will perform integer arithmetic, since *x* is an `int`. This will mean that the result of 3 divided by 2 is 1, with remainder 1 discarded. After this the statement

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

given that *y* is 2 will mean that *z* = 2 x 1 + 3 x 2, *i.e. z* = 2 + 6 = 8.

Hence `System.out.println(z+y);` will output 10, since *z* is 8 and *y* is 2.

Part (iii) was usually answered correctly.

**c. Answer**

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

**Comments**

Many correct answers were seen, but also quite a few incorrect answers. This was quite a challenging question, so those who answered correctly are congratulated.

## Question 4

This was not a popular question, attempted by a little over one third of candidates.

**a. Answer**

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

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

iii. (A) The program will output the unicode value of each character in the file

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

**Comments**

Most candidates attempting the question answered part (a) correctly. Very few wrong answers were seen, and these were usually to part (iii), where some candidates thought that the program would output the text in the file, and others that (D) *None of the above* was the correct answer. This showed that a minority of 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` that had just been read from the file.

**b. Answer**

i. `unable to continue`

ii. `FileNotFoundException`

iii. `NumberFormatException`

**Comments**

Most candidates answered part (i) correctly, but some candidates struggled with parts (ii) and (iii). Many candidates struggled to remember the name of the exception that would be thrown in part (ii), and most struggled to name the exception in part (iii), with some claiming that the expression `Integer.parseInt("five");` would not type check, or would cause a compilation error. Others offered `IOException`, or `InputMismatchException`, which gained no credit, while some credit was given to candidates who wrote such things as *NumbersFormatMismatch* or *FormatNumberException*.

**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**

By far the most common error seen with answers to part (c) was copying the streams to read from the file from part (a). However, the *filey* class from part (a) is reading in from the input file `char` by `char`. What is needed here is to read in a line, and then attempt to parse that line to an `int`. Reading in `char` by `char` will give the Unicode value of each `char` in the file, and adding up those Unicode values will not give the correct answer.

In the above model answer, a `Scanner` has been chained to a `FileReader`. The `FileReader` class reads in streams of characters, while the `Scanner` class has a method, *nextLine()* that reads the input stream and returns the next line of text, excluding the line separator at its end. Since we assume that each line of text in the input file contains a `String` that can be parsed to an `int`, the *readFromFileAndAdd()* method parses each line to an `int`, and adds them to the *sum* variable.

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.

Other errors seen were forgetting to parse each line of input to an `int`, forgetting to close the input file, not writing the method with try/catch blocks

for exception handling, and logical errors with the loop to read from the file, usually meaning that the first or last value read was not included in the addition.

## Question 5

This was not a popular question, attempted by a little under one third of candidates.

**a. Answer**

i. `char`

`int`

ii. (A) When you make an assignment to a reference variable, the variable does not hold the value assigned, but instead points to it in memory. **TRUE**

(B) Variables in Java have unlimited scope, and hence can be accessed from anywhere in their class. **FALSE**

iii. `hello`

iv. `7`

**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 many candidates do not know this simple rule, since so many included *Double* as a primitive variable. It seems that many candidates had not read section 4.4 of Volume 1 of the subject guide which lists the 8 primitive variable types, or section 5.4 of Volume 2 of the guide that discusses simple (primitive) and reference variables.

A few candidates thought that statement (B) in part (ii) was true. Any candidate who thinks that (B) is true is urged to read section 5.3.1 of the second volume of the subject guide, which briefly discusses the scope of variables, which in general is limited to the block of code in which they are declared. For example, instance variables are in scope throughout their containing class, while variables declared in a method are only in scope in that method.

Some candidates thought that the output in part (iii) was `goodbye`, and many thought that the output in part (iv) was `1`. These candidates should note that the *StringParam* class given in part (iii) was taken from exercise 5.7.3 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.

**b. Answer**

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

ii. (D) The *FloatToChar* class will compile and output a `char` (in this case `Z`); the *BooleanToInt* class will not compile.

iii. (B) `121`

`y`

**(b) Comments**

About half of candidates answered part (i) correctly, with the rest believing that either A (*The program will have a run-time error because of a name clash with the two print methods*) or B (*This is an example of method overriding and will not prevent the program from compiling and running*) were the correct answers.

The majority of candidates answered part (ii) incorrectly, usually thinking that the answer was C (*The BooleanToInt class will compile and output 1; the FloatToChar class will not compile*) and sometimes B (*Both classes will not*

*compile*) and less often A (*Both classes will compile*). It is worth noting that this question was taken from the exercises 6.9.3 and 6.9.7 in Volume 2 of the subject guide.

About half of candidates answered part (iii) correctly, with others believing that the correct answer was either A `(8A y)` or C (*No output – compilation error*).

(B) is the correct answer because in the statement `System.out.println('8'+'A');` the single quotes mean that the variables are `chars`. Since a `char` can be an `int` or a character from the keyboard, the JVM treats them as `ints` and adds them. The Unicode value of '8' is 56, and that of 'A' is 65, giving 121. Of course, candidates were not expected to know that the answer would be 121, but to understand that the first line of output would be an integer.

In the second line the statement `System.out.println((char)('8'+'A'));` is casting the `int` value 121 to a `char`. Again 121 comes from the compiler treating '8' and 'A' as `ints`, adding them and getting 121. The result of the cast is 'y'. Again, candidates were not expected to know that 'y' would be the output, but that a `char` would be the output. Since (B) was the only answer with an `int` followed by a `char`, it was the most likely output of the *Chars1* class. It is worth noting that the *Chars1* 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);
}//text in bold given to candidates in the question
```

**Comments**

By far the most common error seen with this question was putting the statement given with the question,

`String encrypted = CaeserCypher.encrypt(text, shift);`

at the start of the method. However, at the start of the method 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. Other basic errors were also seen, such as assigning the values returned from the methods *getTextFromUser()* and *getNumberFromUser()* to variables called, for example, *x* and *y*, but then using *text* and *shift* in the statement given with the question: `String encrypted = CaeserCypher.encrypt(text, shift);` This would lead to compilation errors when the compiler could not find *text* and *shift*. Another basic error was calling the two methods, *getTextFromUser()* and *getNumberFromUser(),* without assigning the values returned by the methods to variables, so that again, the compiler would not be able to find *text* and *shift*.

## Question 6

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

**a. Answer**

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

(B) Constructors have return types **FALSE**

(C) A constructor is used to make an instance of its class **TRUE**

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

(E) The keyword this can be used in a constructor to distinguish the formal parameter from the field name **TRUE**

(F) An instance method does not have the keyword static in its heading **TRUE**

(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 the public instance methods of its parent class **TRUE**

**Comments**

The majority of candidates gave most answers correctly, with many giving entirely correct answers. A few candidates thought that (D) was true, when in fact a class can have an unlimited number of constructors, provided that their parameter lists are all different.

**b. Answer**

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

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

Methods are in bold. 1 & 3 (*isSent*); 8 & 10 (*toString*); and 7 (*printArray*). Their order is arbitrary.

Most 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.    private boolean isSent() {

3.      return cardSent;
      }
```

```
8.      public String toString(){
          String s = ("The birthday of "+name+" is on ");
          s = s+(birthday+".");
          s = s+(" Card sent: "+isSent());
10.       return s;
        }
7.      public static void printArray(Birthday[] b){
          for(int i = 0; i<b.length; i++)
            if (b[i] != null)
              System.out.println(b[i]);
        }
12.     public static void main(String[] args) {
        //test statements
11.       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. There were a few errors, common ones being placing the fragment containing the class closing bracket (fragment 9) somewhere in the body of the class, rather than right at the end. This would also be an error with the main method, as fragment 9 contained one bracket to close the main method, and a second to close the class.

In fact, all errors seen involved the main method in some way. There are 4 fragments that together comprise the main method, and they had to be given in this exact order for full credit: 12, 11, 13, 9. Fragment 12 opens the main method. Fragment 11 makes two new *Birthday* objects and fragment 13 adds the new objects to an `Array` of *Birthday* objects. Fragment 9 then calls the *printArray()* method, in order to print the fields of the new objects, and follows this with two brackets – the first closes the main method and the second closes the class.

A common error was putting fragment 11, comprising the statements that make two new *Birthday* objects, into the constructor. Hence two statements that invoked the constructor were in the constructor. Another common error was 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 quite 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

From reading the *getQuestions()* method in the *PopQuiz* class, candidates were expected to understand that the method was using a two parameter constructor to make the *Question* objects, and that the parameters were a `String` and a `boolean`, in that order. Hence candidates were expected to deduce that the *Question* class needed an instance variable `String` to hold the question, a `boolean` instance variable to hold the answer, and a two parameter constructor: *`Question(String, boolean)`*.

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. The best answers understood that using methods to access the instance variables of the *Question* class, meant that the variables themselves should have *private* access. Making the fields of an object private, and providing access to them through public getter methods is standard practice. However, it was not necessary for candidates to give the instance variables of their *Question* class private access for full credit. Classes whose instance variables had public access, and no access modifier keyword at all, could also gain full credit.

For some reason a few candidates thought that the class should have two String instance variables, but in general this question was either answered correctly by well prepared candidates, or was answered very badly by candidates who did not understand enough to even attempt to write a constructor for the class.