# Examiners' commentaries 2016–17

## CO2220 Graphical object-oriented and internet programming in Java – Zone A

### General remarks

The majority of candidates demonstrated a good grasp of Java; while a small number exhibited an excellent understanding of the subject. However, a significant minority would benefit from more hands-on Java programming practice.

### Comments on specific questions

#### Question 1

This was a popular question, answered by 93 per cent of candidates.

##### Part a

Constructor 1 [88 per cent correctly thought this **would not** compile]

Constructor 2 [80 per cent correctly thought this **would** compile]

Constructor 3 [64 per cent correctly thought this **would** compile]

Constructor 4 [80 per cent correctly thought this **would not** compile]

Constructor 5 [72 per cent correctly thought this **would not** compile]

**Constructor 1**

Half of those who knew that constructor 1 would not compile, understood that it was because the keywords *super* and *this* (when *this* is used to call another constructor in the same class) could not both be used. An equivalent answer accepted by the examiners was that the constructor was calling two other constructors. Some candidates gave no reason for saying that constructor 1 would not compile; while others gave various reasons that were clearly guesses.

**Constructor 2**

Many of those who incorrectly thought the constructor would not compile, thought it was because this and super could not be used in the same constructor. When *this* is used for the purpose of avoiding ambiguity it **can** be used with *super*.

**Constructor 3**

Candidates who incorrectly thought this constructor would not compile gave reasons such as there was no super class constructor with necessary signature (these candidates were clearly misreading the parent class) or that the `int` instance variable was not given a value (true, but not a compilation error).

**Constructor 4**

Of those who correctly answered that constructor 4 would not compile, 90 per cent gave the right reason – because there was no *super* class constructor with the necessary signature; or equivalently, that the variables in the call to super were the wrong way round. The remainder gave no answer, or gave an answer that was clearly a guess by a badly prepared candidate because it bore so little relation to any possible compilation problems.

### Constructor 5

Of those who correctly answered that the constructor would not compile, 56 per cent gave the right reason – because the variable *i* had private access in *Boo* and so could not be accessed. An equivalent answer accepted by the examiners was that i was not defined in *SonOfBoo*. Various incorrect reasons were given, which were probably guesses by candidates picking their third answer at random.

## Part b

In (i) a minority of candidates, 40 per cent, understood that the output would be:

```
Dog name is null
Dog age is 0
```

Candidates generally understood that the program would output:

```
Dog name is X
Dog age is Y
```

Various errors were seen. In place of X candidates wrote various things, nothing at all, 'name', '_'. In place of Y candidates wrote 'zero', 'null' or nothing at all, leaving the end of the sentence blank. These answers showed that many candidates did not understand either or both of the following: (1) that uninitialised instance variables are given default values; and (2) what those default values are.

In (ii) 80 per cent of candidates wrote a correct constructor as follows:

```
public PoorDog (int age, String name){
   this.age = age;
   this.name = name;
}
```

Most incorrect answers were getters or setters for the instance variables instead of a constructor.

In (iii) 48 per cent of candidates correctly explained that adding a constructor to the class had overwritten the default no-argument constructor. Hence the attempt to make an object with no parameters would fail. While 16 per cent of candidates gave an entirely incorrect answer, 36 per cent also gained no credit because they explained that the error was caused by attempting to make a *PoorDog* object with no parameters, when the constructor required two parameters. These candidates did not get any credit because they did not explain why it was that the previous version of the *PoorDog* class successfully compiled and ran with the statement that now caused the error.

### Part c

Only 36 per cent of answers seen were completely correct. The most common reason for losing credit was writing a correct constructor, but one that did not accept two parameters, and did not set the value of the *question* variable using the *super* keyword, as shown in the model answer below:

```
public class FreeQuizQuestion extends Question {
   private String answer;
   public FreeQuizQuestion(String question,String answer){
      super(question);
      this.answer = answer;
   }
   public String getAnswer() {
      return answer;
   }
}
```

## Question 2

This question was attempted by 85 per cent of candidates.

### Part a

Part (i) was answered correctly by 83 per cent of candidates, with the most popular incorrect answer being that the *runAwayFromTrex()* method should have been implemented in class *Q2a*. This incorrect answer was a possible misreading of the question, as it would have been true if the *Q2a* class was extending the *Triceratops* class, rather than trying to make an object of the *Triceratops* class. Hence, the correct answer was that the *Q2a* class would give a compilation error as an abstract class cannot be instantiated.

For part (ii), again, 83 per cent of candidates understood that the *Q2b* class would successfully compile, as abstract classes can have concrete (non-abstract) methods. The most popular incorrect answer was that the class would not compile because of the non-abstract *toString()* method. Sensibly no candidate answered (C), that the class would have a run-time error, since without a main method the class could not be run.

In part (iii) 70 per cent of candidates correctly thought that the *Omnivore* class would compile. Candidates who thought that the *Omnivore* class would not compile did not understand that as an abstract class it did not have to implement the abstract methods of the *Mammal* interface. Any concrete sub-classes of Omnivore would have to implement the *Mammal* interface's abstract methods.

Of those attempting the question, 74 per cent correctly thought that the Rabbit class would compile. Candidates who thought that the *Rabbit* class would not compile failed to recognise that as the class implemented both the abstract methods of the *Mammal* interface and was syntactically correct, it would compile successfully.

The majority of candidates attempting the question, 65 per cent, correctly understood that the *Marsupial* class would not compile as it did not implement all the abstract methods of the *Mammal* interface that it implemented.

#### Part b

In part (i) 70 per cent understood that the statement `String s = a.get(0);` would give the 'Object cannot be converted to String' error. Candidates should have demonstrated their understanding that since the `ArrayList` was not parameterized, anything stored in it would be an `Object`, and would need to be cast to a type when being retrieved from the `ArrayList`. Hence, the correction needed was `String s = (String) a.get(0);`

Some candidates stated the correction was to parameterize the arraylist with `<String>`. While this would have dispensed with the error, candidates were asked to identify the statement giving the error, which was the `String s = a.get(0);` statement, as the compiler would interpret this as an attempt to assign an `Object` to a `String` variable. The examiners gave some credit for this type of answer.

In part (ii) 74 per cent of candidates correctly gave the statement causing the compilation error as `b.add(new Machine());` This was because the compiler would only allow Computer objects and any sub-types to be inserted into `ArrayList` *b*. *Machine* is not a sub-type of *Computer so Machine* objects cannot be inserted into `ArrayList` *b*.

Some candidates thought that the statement giving the error was

```
ArrayList <Computer> b =new ArrayList<Computer>();
```

These candidates suggested that the `ArrayList` should be of type *Machine*. Again, this showed some understanding and gained partial credit.

In (iii) the following answers were correct:

(A) TRUE [83 per cent answered correctly]

(B) TRUE [78 per cent answered correctly]

(C) FALSE [70 per cent answered correctly]

## Part c

In part (c) 83 per cent of candidates gave a completely correct answer. One common mistake was putting the *howl()* method in the *Labrador* class, which meant that both *Dog* and *Labrador* would compile, but the *NoisyDog* class would not. Another common error was placing the class closing bracket for *Dog*, statement /*8*/, at the end of *Labrador*, making *Labrador* an inner class of *Dog*. This would mean that *NoisyDog* would not compile as it would not be able to find *Labrador*. Some candidates also placed *Dog's* instance variables after its constructor, which would work but was not the order that candidates were asked to follow. More serious errors were placing the instance variables in the *Labrador* class, which would give compilation errors, and confusing the statements that should have been inside the constructors of *Dog* and *Labrador*.

```
/*3*/ public class Dog{
   /*13*/    private String name;
             private boolean oldDog;
   /*10*/    public Dog (String name, boolean isOld){
   /*11*/           this.name = name;
                  oldDog = isOld;
             }
   /*7*/     public boolean getOldDog(){
                  return oldDog;
             }
   /*4*/     public String getName(){
                  return name;
             }
   /*5*/     public void howl (){
                  System.out.println("ooooooooooaaaawooool");
             }
   /*14*/    public void growl(){
                  System.out.println("grrrrr");
             }
   /*8*/ }
   //the order of the four instance methods is arbitrary.
   /*1*/ public class Labrador extends Dog{
   /*9*/     public Labrador (String name, boolean old){
   /*12*/    super(name, old);
             }
   /*6*/     public void growl(){
             super.growl();
   /*2*/     System.out.println("grrr grrrrr GRRRRRRR");
             }
      }
```

In (iii) the following answers were correct:

(A) TRUE [83 per cent answered correctly]

(B) TRUE [78 per cent answered correctly]

(C) FALSE [70 per cent answered correctly]

In part (c) 83 per cent of candidates gave a completely correct answer. One common mistake was putting the *howl()* method in the *Labrador* class, which meant that both *Dog* and *Labrador* would compile, but the *NoisyDog* class would not. Another common error was placing the class closing bracket for *Dog*, statement /*8*/, at the end of *Labrador*, making *Labrador* an inner class of *Dog*. This would mean that NoisyDog would not compile as it would not be able to find *Labrador*. Some candidates also placed *Dog's* instance variables after its constructor, which would work but was not the order that candidates were asked to follow. More serious errors were placing the instance variables in the *Labrador* class, which would give compilation errors, and confusing the statements that should have been inside the constructors of *Dog* and *Labrador*.

The correct answer follows:

```
/*3*/       public class Dog {
/*13*/              private String name;
                    private boolean oldDog;
/*10*/              public Dog (String name, boolean isOld){
/*11*/                      this.name = name;
                          oldDog = isOld;
                    }
/*7*/       public boolean getOldDog(){
            return oldDog;
            }
/*4*/       public String getName(){
            return name;
            }
/*5*/       public void howl (){
                    System.out.println("ooooooooaaaawooool");
            }
/*14*/      public void growl(){
                    System.out.println("grrrrr");
            }
/*8*/ }
//the order of the four instance methods is arbitrary.
/*1*/       public class Labrador extends Dog{
/*9*/               public Labrador (String name, boolean old){
/*12*/                      super(name, old);
                    }
/*6*/               public void growl(){
                          super.growl();
/*2*/                       System.out.println("grrr grrrrr
                          GRRRRRRR");
                    }
            }
```

Candidates should have been able to answer this question by seeing from the code fragments that *Labrador* extends *Dog*. They should have realised that the statement `super(name, old);` is a call to a superclass constructor, and thus must belong to the Labrador constructor. Since there are only two instance variables, they must belong to the Dog class in order to make the *Labrador* constructor compile. This should have helped candidates understand what statements must belong to the *Dog* constructor.

The methods *getOldDog()* and *getName()* then have to belong to *Dog* since the variables they are providing access to have private access in *Dog*, meaning they would not be legal in the *Labrador* class. Candidates should have understood from *NoisyDog* that *howl()* must belong to the *Dog* class, as it is referenced by a *Dog* object. From *NoisyDog* they should also understand that there are two *growl()* methods, therefore there must be one in each class. The fragment containing the statement `super.growl();` must belong to the growl() method in *Labrador*, as it is calling the superclass *growl()* method. Candidates should have been able to complete *Labrador's growl()* method by considering the *NoisyDog* output.

## Question 3

This question was attempted by 22 per cent of candidates.

### Part a

| | | |
|---|---|---|
| (A) | TRUE | [100 per cent answered correctly] |
| (B) | TRUE | [50 per cent answered correctly] |
| (C) | FALSE | [17 per cent answered correctly] |
| (D) | TRUE | [50 per cent answered correctly] |
| (E) | TRUE | [75 per cent answered correctly] |
| (F) | TRUE | [83 per cent answered correctly] |
| (G) | FALSE | [83 per cent answered correctly] |
| (H) | TRUE | [67 per cent answered correctly] |

### Part b

(i) All candidates understood that the corner squares, and the square in the middle were filled with the *color2* variable, and the alternating squares with *color1*.

(ii) Eighty-three per cent knew the correct answer was statements 9 and 10.

(iii) All candidates correctly gave (B) as the answer, understanding that the program had been written such that increasing the size of the frame while the program was running would mean that the rectangles would scale with the frame.

(iv) Fifty per cent of candidates gained full credit for this part of the question. Candidates lost marks for failing to make a random *Color*; for marking a random Color but failing to update one of the variables with it as appropriate; and for correct logic but very poor syntax.

A correct answer follows. Note that the answer uses the *color1Change* `boolean` instance variable of the *NineRectanglesGUI* class. Only a third of candidates answering the question realised that the `boolean` instance variable could be used in their answer.

```
class RColorListener implements ActionListener{
   public void actionPerformed (ActionEvent e){
      int r = (int)(Math.random()*255);
      int gr = (int)(Math.random()*255);
      int b = (int)(Math.random()*255);
      if (color1Change) color1 = new Color(r,gr,b);
      else color2 = new Color(r,gr,b);
      color1Change = !color1Change;
      dP.repaint();
   }
}
```

## Question 4

This question was attempted by 85 per cent of candidates.

### Part a

In part (i) 83 per cent of candidates understood that statement (A) was true and that statement (B) was false.

In part (ii) 85 per cent of candidates understood that statement (A) was the missing statement. It should have been clear to candidates that either (A) or (B) had to be the missing statement, as both gave 3 spaces to the `int` variable, which meant that it was output as '011'. Candidates would then have understood that statement (B) was giving 28 spaces to display `String` *a*, meaning that if this was the missing statement there would be a large gap between "Come in number" and 011. Therefore statement (A) could have been chosen by a process of elimination. Despite this, 17 per cent thought that statement (C) was the correct answer.

In part (iii) 78 per cent of candidates understood that the correct output was given by (B), with most wrong answers being (C).

### Part b

In (i) 70 per cent correctly answered that the *FORINSTANCE* variable had not been initialised as `static final` variables must be. Various wrong answers were seen, without any common errors since candidates were clearly guessing, writing such things as '*a variable cannot be both static and final*' and '*a new value cannot be added after the final int variable*'.

In (ii) 87 per cent of candidates understood that *AskToDance()* in *Happier* overrides *askToDance()* in *Happy*. Since a*skToDance()* is a final method, it cannot be overridden and the compiler will flag this as an error.

In (iii) 87 per cent of candidates understood that the variable area was final and hence could not have its value changed by the *calcArea()* method.

### Part c

A minority of candidates, 48 per cent, gained full credit for this question. As many as 30 per cent of candidates received no credit, as they clearly had no idea how to attempt a sensible answer. Those candidates who received partial credit generally started well by using *String.split()* to make an array from the *line* variable. They then lost marks by failing to parse the last but one item in the array to an `int`, and/or for not writing some logic to take the value of the final item in the array and use it to work out how to give the correct value to the `boolean` *permanent* field of the *Employee* object.

The following is an example of how to write a correct *parseEmployee(String)* method. Most of the method is standard, with only the logic of working out whether the boolean variable should be true or false amenable to other approaches:

```
private static Employee parseEmployee(String line) {
   String[] details = line.split(",");
   String name = details[0];
   String jobTitle = details[1];
   String teamName = details[2];
   int age = Integer.parseInt(details[3]);
   int temp = Integer.parseInt(details[4]);
   boolean permanent = true;
   if (temp == 0) permanent = false;
   return new Employee(name, jobTitle, teamName, age,
      permanent);
}
```

## Question 5

This question was attempted by 78 per cent of candidates.

### Part a

(i) Fifty-seven per cent of candidates correctly answered that the other type of stream is *chain*. Some candidates skipped this question; others gave guesses such as 'buffered'; 'Thread'; 'IOStream'; and 'file'.

(ii) Niney per cent of candidates correctly answered that the second statement, using `BufferedWriter`, would be the best to use in terms of efficient use of system resources. Only 62 per cent could explain that this was because using a `BufferedWriter` is more efficient, as it fills its buffer, and does not write to the file until the buffer is full. This means less writing to the file and so less overhead as manipulating data in memory is faster.

(iii) Ninety per cent of candidates understood that the correct answer was:

(B) The class will print the IP address of the current machine to standard output.

Statements (C) and (D) were both incorrectly given, with no candidate choosing (A).

### Part b

(i) The correct answer, given by 81 per cent of candidates, was an object's state, given by its instance variables is saved. Any objects that are referenced by the instance variables, and in turn any further objects referenced by their instance variables, etc. are also saved. The most popular incorrect answer chosen was (B), given by 14 per cent of candidates who thought that static variables are saved when objects are serialized. Fortunately, very few candidates answered: (C) *The object's source code*, since that answer is clearly nonsense.

(ii) Correct answers were as follows:

| | |
|---|---|
| (A) FALSE | [52 per cent answered correctly] |
| (B) TRUE | [71 per cent answered correctly] |
| (C) TRUE | [76 per cent answered correctly] |
| (D) TRUE | [67 per cent answered correctly] |

(iii) The correct answer, given by 81 per cent of candidates, was (A), that one issue with object deserialization is a possible change of class definition, which may break the deserialization process. A serial version ID can solve this problem as it enables the JVM to assess if the class is compatible with the serialized object. Fourteen per cent of those answering thought that (C) was the correct answer with the remainder choosing (B).

## Part c

No candidate gained full marks for this question; 57 per cent gained partial credit with the remainder either making no attempt (10 per cent); or receiving no credit (33 per cent) for copying their answer from the *serializeToFile(Point)* method given with the question, and simply changing 'Output' to 'Input'.

An example of a method that would have achieved full credit:

```
public static Point deserializeFromFile(){
   Point temp = new Point();
   try {
      FileInputStream fis = new FileInputStream(s);
      ObjectInputStream ois = new
      ObjectInputStream(fis);
      temp = (Point)ois.readObject();
      fis.close();
      ois.close();
   }
   catch (Exception ex) {
      ex.printStackTrace();
   }
   return temp;
}
```

There are three potential exceptions that could be thrown by a correctly written method: `FileNotFoundException`, `IOException`, and `ClassNotFoundException`. The `ClassNotFoundException` is the possible result of attempting to cast the deserialized object to a *Point* object; this exception cannot be caught by `IOException` as it is not a child class of `IOException`. Therefore, for full credit, candidates should have explicitly caught the `ClassNotFoundException` or included an `Exception` catch block with the try block encompassing the attempt to cast to *Point*. In fact, only 28 per cent of candidates handled the `ClassNotFoundException`; this was the most common mistake. Most candidates included only an `IOException` catch block, probably copying from the *serializeToFile(Point)* method.

Candidates should have understood from the main method statement: `b=deserializeFromFile();` that the *deserializeFromFile()* method must return a *Point* object. Therefore for full credit, the method written should have both cast the deserialized object to *Point* and returned a *Point* object (i.e. the method should have been of type *Point*). In fact, 19 per cent of candidates made a *Point* object but did not attempt to cast the deserialized object, a compilation error. Another 19 per cent made no *Point* object at all, deserializing to `Object` (or in a few cases to `String`), and writing void methods.

## Question 6

This question was attempted by 37 per cent of candidates.

### Part a

(i) The correct answer, given by 90 per cent of candidates, was that the output was `Tealc`, although a minority incorrectly thought it was `Vala`.

(ii) The correct answer, given by 60 per cent of candidates, was that all exceptions are sub-classes of the *Exception* class. Incorrect answers given included *Throwable* (given by 30 per cent) and Runnable.

(iii) The correct answer, (A) was given by 80 per cent of candidates, with the remainder choosing (C).

(iv) Most candidates (90 per cent) understood that `ArithmeticException` and `ArrayIndexOutOfBoundException` are unchecked exceptions, with 60 per cent understanding that `NumberFormatException` is also unchecked.

Most candidates (90 per cent) understood that `UnknownHostException` is a checked exception, with 80 per cent understanding that so is `FileNotFoundException`.

### Part b

(i) The correct missing word for statement (1), given by 60 per cent of candidates, was BLOCKED. Incorrect answers seen included *running*, *dropped* and *interrupted*.

The correct missing word from statement (2), NEW, was given by 20 per cent of candidates, with various wrong answers seen, such as *open*, *run*, *runnable* and *sleeping*.

(ii) The correct answer, given by 90 per cent of candidates, was (C), the 'lost update' problem in thread programming is when a thread 'wakes up' and continues operating on a value that it had read before going to sleep, not knowing that another thread has changed it. Statement (A) was the answer incorrectly chosen by 10 per cent of candidates.

(iii) Correct answers to the true or false questions posed were:

| | |
|---|---|
| (A) TRUE | [100 per cent answered correctly] |
| (B) FALSE | [10 per cent answered correctly] |
| (C) TRUE | [100 per cent answered correctly] |
| (D) FALSE | [20 per cent answered correctly] |

### Part c

Only 20 per cent of candidates attempting this question achieved full credit. The remainder achieved partial credit, except for the 20 per cent who made no attempt at part (c).

Candidates were asked to add exception handling to the *SourceViewer* class, and instructed to include at least two *catch* blocks in their exception handling. Choice of the exceptions to handle was where most candidates lost credit, although a few candidates made other mistakes with their exception handling.

A possible correct answer would be as follows (text that is part of the answer is indicated in bold):

```
import java.io.*;
import java.net.*;
public class SourceViewer1{
   public static void main(String args []){
      try {
         URL u = new URL(args [0]);
         Reader r = new BufferedReader
         (new InputStreamReader(u.openStream()));
         int c = -1;
         while ((c = r.read())!= -1) {
            System.out.print((char)c);
         }
      }
      catch (MalformedURLException e) {
         System.err.println(e);
      }
      catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

To achieve full credit candidates had to explicitly handle the potential `MalformedURLException` thrown by the *URL* object, as well as the possible `IOException`, which could also be caught using `Exception`.

Therefore, for full marks, catch blocks could have been one of the following combinations:

- `MalformedURLException; Exception.`
- `MalformedURLException; IOException.`
- `MalformedURLException; IOException; Exception.`

Candidates lost marks by including only one catch block or by including catch blocks for exceptions that could not be thrown by the *SourceViewer* class, such as `ArrayIndexOutOfBoundsException`. Candidates also lost marks by including two catch blocks, both of them for `IOException`.

Candidates were given some credit for attempting to handle the `MalformedURLException` but without being able to remember its name, calling the exception *UnresolvableURLException* or *NetException*, for example.

Mistakes made with exception handling were:

- starting the try block after the `URL u = new URL(args [0]);` statement, meaning that the compilation error: `unreported exception MalformedURLException;` must be caught or declared to be thrown would be produced;
- putting the `MalformedURLException` catch block after the `IOException` catch block; a compilation error since the order of catch blocks matters. The compiler visits them in order, and the `MalformedURLException` would be dealt with by the `IOException`, making the `MalformedURLException` catch block redundant, which the compiler would flag `(error: exception MalformedURLException has already been caught).`

## Conclusion

The majority of candidates demonstrated a good grasp of Java, with a minority demonstrating an excellent understanding of the subject. A significant minority were not well prepared, and would benefit from more hands-on programming practice.

# Examiners' commentaries 2016–17

## CO2220 Graphical object-oriented and Internet programming in Java – Zone B

### General remarks

Most candidates demonstrated a good grasp of Java; with a small number demonstrating an excellent understanding of the subject. A significant minority would benefit from more hands-on Java programming practice. Some candidates may have improved their mark with a better standard of written English.

### Comments on specific questions

#### Question 1

This was a popular question, attampted by 87 per cent of candidates.

##### Part a

Constructor 1 [55 per cent correctly thought this **would not c**ompile]

Constructor 2 [65 per cent correctly thought this **would** compile]

Constructor 3 [72 per cent correctly thought this **would** compile]

Constructor 4 [93 per cent correctly thought this **would not** compile]

Constructor 5 [93 per cent correctly thought this **would not** compile]

Candidates achieved the credit for this answer if they correctly identified both the constructor, and the reason for the compilation error.

**Constructor 1**

Only 36 per cent of those who knew constructor 1 would not compile understood that it was because the keywords *super* and *this* (when *this* is used to call another constructor in the same class) could not both be used in the same constructor. Candidates focused on the `this(s);` statement as the reason for the error, writing such things as:

- invalid `this(s)` because *SonOfWoo* has no `String` instance variable
- `this(s)` doesn't refer to anything (assuming only one constructor in *SonOfWoo*)
- `this(s)` is not a legal command in Java
- `this(s)` should be `this.s`

Clearly many candidates did not understand that `this(s);` represented a call to another constructor in the same class that took a single `String` parameter.

**Constructor 2**

A few of those who incorrectly thought the constructor would not compile thought it was because *this* and *super* could not be used in the same constructor. When *this* is used for the purpose of avoiding ambiguity it can be used with *super*. Apart from this, there were no common reasons given for believing the constructor would fail to compile.

**Constructor 3**

Quite a number of the candidates who incorrectly thought this constructor would not compile related their reasoning to the constructor's parameters, writing that '*the constructor has an argument it does not use*' (true but not a compilation error); or, the most popular choice, '*constructor 3 takes a* `String` *parameter and calls super with an* `int`.'

These candidates did not understand that the parameters (or arguments) given to constructors do not affect compilation as long as they are variable types the compiler recognises. The compiler does not care if the parameter is not actually used, or if neither the current class or any of its parent classes have an instance variable of that type.

**Constructor 4**

Of those who correctly answered that constructor 4 would not compile, 73 per cent gave the right reason – because there was no super class constructor with the necessary signature, or equivalently, that the variables in the call to *super* were the wrong way round; the remainder gave no answer or gave various wrong answers that bore little or no relation to actual compilation errors.

**Constructor 5**

Of those who correctly answered that the constructor would not compile, 68 per cent gave the right reason – because the variable s had private access in *Woo* and so could not be accessed. The 68 per cent include those who gave one of the various equivalent answers that were seen and accepted by the examiners. These included that s was not defined in *SonOfWoo*; or was not a *SonOfWoo* instance variable; and that s was not declared as a parameter to the constructor.

Those candidates who gave the wrong reason focused on the constructor's parameters, stating that the constructor's `boolean` parameter was not assigned to anything; or that the parent class did not have a constructor with a `boolean` parameter. Some stated simply that the `boolean` parameter was an error with no further reason given; while others thought that the parameter given to the constructor should match the parameter in the call to *super*.

## Part b

In (i) a minority of candidates, 47 per cent, understood that the output would be:

```
Dog name is null
Dog age is 0.0
```

The most popular wrong answer, given by 20 per cent of candidates was:

```
Dog name is
Dog age is
```

Other mistakes were writing '*Dog name is name*' and '*Dog age is age*'; or failing to appreciate that the output of the age variable would be 0.0, and not '0'.

In (ii) 90 per cent of candidates wrote a correct constructor as follows:

```
public PoorDog (double age, String name){
   this.age = age;
   this.name = name;
}
```

In (iii) 52 per cent of candidates correctly explained that adding a constructor to the class had overwritten the default no-argument constructor. Hence, the attempt to make an object with no parameters would fail. While 18 per cent of candidates gave either no answer or an entirely incorrect answer, 30

per cent gained no credit because they explained that the error was caused by attempting to make a *PoorDog* object with no parameters, but did not explain why it was that the previous version of the *PoorDog* class successfully compiled and ran with the statement that now caused the error.

### Part c

Only 35 per cent of answers seen were completely correct. The most common reason for losing credit was writing a correct constructor, but one that did not accept two parameters, and/or did not set the value of the *question* variable using the *super* keyword to call the parent class constructor, as is done in the model answer below:

```
public class TrueFalseQuestion extends Question {
   private boolean answer;
   public TrueFalseQuestion(String question, boolean answer) {
      super(question);
      this.answer = answer;
   }
   public boolean getAnswer() {
      return answer;
   }
}
```

Another reason for losing credit was including the `String` question instance variable in the *TrueFalseQuestion* class.

## Question 2

This question was attempted by 98 per cent of candidates.

### Part a

Part (i) was answered correctly by 80 per cent of candidates who wrote that an abstract class cannot be instantiated. Incorrect answers focused on the *roar()* method, giving such answers as '*the class will not compile because of the roar() method*', and '*because the roar() method is abstract*', or '*because roar() cannot be extended*' and '*because of an abstract variable in a non-abstract class*'.

For part (ii) 64 per cent of candidates recognised that (A) was the correct answer; namely, the Q2b interface would not compile since all methods in an interface must be abstract. A large minority, 22 per cent, thought that the interface would compile successfully; while 9 per cent of candidates thought that the interface would compile but the non-abstract method would give a run-time error in implementing classes.

In part (iii) just 31 per cent of candidates correctly thought that the *Nocturnal* class would compile. Candidates who incorrectly thought the class would not compile did not understand that since *Nocturnal* was an abstract class, it did not have to implement *Mammal's* abstract methods, but could leave that to any extending concrete classes.

A majority, 73 per cent, understood that the concrete *Monotreme* class would not compile since it did not implement both of the abstract methods from the *Mammal* interface.

Of those attempting the question, 87 per cent correctly thought that the *Rabbit* class would compile. Candidates who thought that the *Rabbit* class would not compile failed to recognise that as the class implemented both the abstract methods of the *Mammal* interface and was syntactically correct, it would compile successfully.

## Part b

In part (i) 73 per cent gave the correct statement that produced the compilation error, but only 42 per cent of these candidates gave the right correction.

The statement that the compiler would identify as giving the 'Object cannot be converted to int' error was `int i = a.get(0);`.

The correction needed was `int i = (int)a.get(0);` That is because a variable retrieved from a non-parameterized `ArrayList` is an object, and needs to be cast to its actual type, or one of its sub-types where appropriate. Most mistakes seen in giving this correction were where candidates recognised the need to convert the object to an `int` variable, but were not sure how, writing such things as:

- `Integer.intValue(a.get(0));`
- `int.Value(a.get(0));`
- `(a.get(0)).intValue();`
- `Integer.parseInt(a.get(0));`

Eighteen per cent of candidates stated that the error was:

```
ArrayList a = new ArrayList();
```

and the correction was:

```
ArrayList<Integer> a = new ArrayList<Integer>();
```

The examiners accepted this answer and gave some credit for it. However, no credit was given where candidates gave the correction as:

```
ArrayList<int> a = new ArrayList<int>();
```

since `ArrayLists` cannot be parameterized with primitive variables. Using `Integer` would work because of wrapping.

In part (ii) 89 per cent of candidates correctly gave the statement causing the compilation error as `b.add(new Vehicle());` The compiler will only allow *Bicycle* objects and any sub-types, to be inserted into arraylist *b*. *Vehicle* is not a sub-type of *Bicycle* so *Vehicle* objects cannot be inserted into *b*.

The examiners did not ask for the reason that the statement produced a compilation error; however, where candidates stated that the error was the statement making `ArrayList` *b*, as the `ArrayList` should be parameterized to *Vehicle*, the examiners gave some credit, since while the compiler will not flag this statement as an error, it is the cause of the `b.add(new Vehicle());` statement being flagged as an error.

In (iii) the following answers were correct:

| | | |
|---|---|---|
| (A) | TRUE | [82 per cent answered correctly] |
| (B) | TRUE | [76 per cent answered correctly] |
| (C) | FALSE | [71 per cent answered correctly] |

## Part c

In part (c) 80 per cent of candidates gave a completely correct answer. One common mistake was putting the *howl()* method in the *Labrador* class, which meant that both *Dog* and *Labrador* would compile, but the *NoisyDog* class would not. Another common error was placing the class closing bracket for *Dog*, statement /*4*/, at the end of *Labrador*, making *Labrador* an inner class of *Dog*. This would mean that *NoisyDog* would not compile as it would not be able to find *Labrador*. A related error was not using statement /*4*/ at all, so that *Dog* had no class closing bracket.

**The correct answer follows:**

```
/*1*/ public class Dog{
/*10*/      private String name;
            private boolean oldDog;
/*8*/       public Dog (String name, boolean isOld){
/*9*/       this.name = name;
            oldDog = isOld;
            }
/*5*/       public boolean getOldDog(){
            return oldDog;
            }
/*12*/      public String getName(){
                  return name;
            }
/*13*/      public void howl (){
                  System.out.println("ooooooooaaaawooool");
            }
/*6*/       public void growl(){
                  System.out.println("grrrrr");
            }
/*4*/}
//order of the four instance methods is arbitrary.
/*3*/ public class Labrador extends Dog{
/*14*/      public Labrador (String name, boolean old){
/*11*/            super(name, old);
            }
/*7*/       public void growl(){
                  super.growl();
/*2*/               System.out.println("grrr grrrrr GRRRRRRR");
            }
     }
```

Candidates should have been able to answer this question by seeing from the code fragments that *Labrador* extends *Dog*. They should have realised that the statement `super(name, old);` is a call to a superclass constructor, and thus must belong to the *Labrador* constructor. Since there are only two instance variables, they must belong to the *Dog* class in order to make the *Labrador* constructor compile. This should have helped candidates to understand which statements must belong to the Dog constructor.

The methods *getOldDog()* and *getName()* then have to belong to Dog since the variables they are providing access to have private access in *Dog*, meaning they would not be legal in the *Labrador* class. Candidates should have understood from *NoisyDog* that *howl()* must belong to the *Dog* class, as it is referenced by a *Dog* object. From *NoisyDog* they should also understand that there are two *growl()* methods; therefore there must be one in each class. The fragment containing the statement `super.growl();` must belong to the *growl()* method in *Labrador*, as it is calling the superclass *growl()* method. Candidates should have been able to complete *Labrador's growl()* method by considering the *NoisyDog* output.

## Question 3

This question was attempted by 15 per cent of candidates.

### Part a

| | | |
|---|---|---|
| (A) | TRUE | [100 per cent answered correctly] |
| (B) | TRUE | [100 per cent answered correctly] |
| (C) | FALSE | [86 per cent answered correctly] |
| (D) | FALSE | [86 per cent answered correctly] |
| (E) | FALSE | [71 per cent answered correctly] |
| (F) | FALSE | [100 per cent answered correctly] |
| (G) | FALSE | [100 per cent answered correctly] |
| (H) | TRUE | [100 per cent answered correctly] |

### Part b

(i) All candidates understood that the corner squares, and the square in the middle were filled with the *color1* variable, and the alternating squares with *color2*.

(ii) All candidates knew the correct answer was statements 17 and 18.

(iii) Forty-three per cent of candidates correctly gave (B) as the answer, understanding that the program had been written such that decreasing the size of the frame while the program was running would mean that the rectangles would scale with the frame, and so the user would always be able to see nine rectangles. Another 43 per cent mistakenly thought that statement (A) was the correct answer (the squares would stay the same size as the frame was reduced). The remaining candidates thought that the user would not be able to shrink the frame while the program was running.

(iv) Fourteen per cent of candidates gained full credit for this part of the question. Many candidates lost marks for failing to recognise that using a local variable to decide which of the `Color` variables to change will not work, as the value of the local variable will be overwritten with every iteration of the program. Other common errors were syntactical; namely, getting the syntax to change the `Color` variable badly wrong; for example, by trying to use a method that does not exist; or failing to write statements that would successfully make the random `Color` variable.

A few candidates had logic in the *actionPerformed(ActionEvent)* method to choose randomly which `Color` to change; the examiners gave credit for this answer, assuming that it was a misread, and recognising that it did not make the answer any easier to write successfully.

A correct answer follows. Note that the answer uses the *color2Change* `boolean` instance variable of the *NineRectanglesGUI* class. Only 40 per cent of candidates answering the question realised that the `boolean` instance variable could be used in their answer.

```
class RColorListener implements ActionListener{
   public void actionPerformed (ActionEvent e){
      int r = (int)(Math.random()*255);
      int gr = (int)(Math.random()*255);
      int b = (int)(Math.random()*255);
      if (color2Change) color2 = new Color(r,gr,b);
      else color1 = new Color(r,gr,b);
      color2Change = !color2Change;
      dP.repaint();
   }
}
```

## Question 4

This question was attempted by 85 per cent of candidates.

### Part a

In part (i) 85 per cent of candidates understood that statement (A) was true and 69 per cent that statement (B) was also true.

In part (ii) 95 per cent of candidates understood that statement (C) was the missing statement. It should have been clear to candidates that either (C) or (D) had to be the missing statement, as both gave four spaces to the `int` variable, which meant that it was output as '0011'. Candidates would then have understood that statement (D) was giving 28 spaces to display `String` a, meaning that if this was the missing statement there would be a large gap between "Come in number" and 0011. Therefore statement (C) was the correct answer. Despite this, incorrect answers were evenly divided between statements (A) and (B).

In part (iii) 90 per cent of candidates understood that the correct output was given by (A), with all incorrect answers being (C).

### Part b

In (i) 82 per cent correctly answered that the *EXAMPLE* variable had not been initialised as `static final` variables must be. Various wrong answers were seen, without any common errors since candidates were clearly guessing, writing such things as '*a final object should not be static*' and '*name and n are null*'.

In (ii) 97 per cent of candidates understood that Variable *circumference* was final and could not be given a value by the *calcCircumference* method. One wrong answer given was that a final method cannot return a value; the actual purpose of marking a method as final is to guarantee that it cannot be overridden.

In (iii) 64 per cent of candidates understood the *Good* class was final and could not be extended by class *Better*. One very popular incorrect answer was to write that *Better* could not override the *greeting()* method as *Good* is final. In fact, this is the error that compilation would produce:

```
1: error: cannot inherit from final Good
public class Better extends Good{
                            ^
1 error
```

Another incorrect answer was that the `super(n);` statement would give the compilation error since the super class constructor could not be called by class *Better*.

### Part c

A minority of candidates, one third, gained full credit for this question; 15 per cent of candidates received no credit, as they made no attempt or provided an answer with so many mistakes that no credit could be given. Those candidates who received partial credit generally started well by using *String. split()* to make an array from the *line* variable. They then lost marks by failing to parse the last but one item in the array to an `int`, and/or for not writing some logic to take the value of the final item in the array and use it to work out how to give the correct value to the `boolean` *permanent* field of the *Employee* object. Another common error was including a `for` or `while` loop in the *parseEmployee(String)* method. Candidates should have been able to understand from reading the *PersonnelFromFile* class and its *readEmployees()*

method that the *parseEmployee(String)* method was expected to take one line of the input file and return one *Employee* object. There was no reason to include a loop in the method and candidates who did lost credit for poor logic.

The following is an example of how to write a correct *parseEmployee(String)* method. Most of the method is standard, with only the logic of working out whether the `boolean` variable should be true or false amenable to other approaches:

```
private static Employee parseEmployee(String line) {
   String[] details = line.split(",");
   String name = details[0];
   String jobTitle = details[1];
   String teamName = details[2];
   int age = Integer.parseInt(details[3]);
   int temp = Integer.parseInt(details[4]);
   boolean permanent = true;
   if (temp == 0) permanent = false;
   return new Employee(name, jobTitle, teamName, age,
      permanent);
   }
}
```

## Question 5

This question was attempted by 67 per cent of candidates.

### Part a

(i) Thirty-five per cent of candidates correctly answered that the other type of stream is *connection*. Many and various incorrect answers were seen including: ''byte'; 'buffered'; 'unchained'; 'non-chain'; and 'InputOutput'.

(ii) Eighty-seven per cent of candidates correctly answered that the second statement, using `BufferedReader`, would be the best to use in terms of efficient use of system resources. Only 41 per cent of those giving the correct answer could explain that this was because using a `BufferedReader` is more efficient, as it reads from the file, filling its buffer, and does not read again until the buffer is empty. This means less reading from the file and so less overhead as manipulating data in memory is faster.

Many candidates lost credit by asserting that using a `BufferedReader` would be more efficient, or a better use of memory, or would use less resources, without explaining why.

(iii) 26 per cent of candidates understood that the correct answer was:

(A) The class will find an IP address from a host name.

Statement (B) was given by 48 per cent of candidates, and statement (D) by 19 per cent, with the remainder giving statement (C).

### Part b

(i) The correct answer, given by 94 per cent of candidates, was an object's state, given by its instance variables is saved. Any objects that are referenced by the instance variables, and in turn any further objects referenced by their instance variables, etc. are also saved. The most popular incorrect answer chosen was (B), meaning that a small minority thought that static variables are saved when objects are serialized.

(ii) Correct answers were as follows:

(A) TRUE [71 per cent answered correctly]

(B) FALSE [03 per cent answered correctly]

(C) TRUE [71 per cent answered correctly]

(D) TRUE [68 per cent answered correctly]

(iii) The correct answer, given by 52 per cent of candidates, was (A), Serial version ID. 35 per cent of those answering thought that (B) was the correct answer, 8 per cent thought (C) with the remainder choosing (D).

## Part c

Only 10 per cent gained full marks for this question; 52 per cent gained partial credit with the remainder either making no attempt (6 per cent) or receiving no credit (32 per cent) for copying their answer from the *serializeToFile(Point)* method given, and simply changing 'Output' to 'Input'.

Candidates lost credit in their answers with their exception handling, making the method void (the method should be of type *Point* so that a *Point* object can be returned), failing to make a *Point* object, failing to cast the deserialized object appropriately and not returning the deserialized object.

An example of a method that would have achieved full credit:

```
public static Point deserializeFromFile(){
   Point temp = new Point();
   try {
      FileInputStream fis = new FileInputStream(s);
      ObjectInputStream ois = new
      ObjectInputStream(fis);
      temp = (Point)ois.readObject();
      //casting the deserialized object to Point
      fis.close();
      ois.close();
   }
   catch (Exception ex) {
      ex.printStackTrace();
   }
   return temp;
}
```

There are three potential exceptions that could be thrown by a correctly written method: `FileNotFoundException`, `IOException`, and `ClassNotFoundException`. The `ClassNotFoundException` is the possible result of attempting to cast the deserialized object to a *Point* object; this exception cannot be caught by `IOException` as it is not a child class of `IOException`. Therefore, for full credit, candidates should have explicitly caught the `ClassNotFoundException` or included an `Exception` catch block with the try block encompassing the attempt to cast to *Point*. In fact only 10 per cent of candidates handled the `ClassNotFoundException;` hence this was the most common mistake.

Candidates should have understood from the main method statement: `b=deserializeFromFile();` that the *deserializeFromFile()* method must return a *Point* object. Therefore for full credit, the method written should have been of type *Point*, and should have both cast the deserialized object to *Point* and returned a *Point* object. In fact nearly half of methods seen were void, and

only 55 per cent of candidates made a *Point* object. Some candidates who did make a *Point* object lost further credit by not returning the deserialized object and/or by not explicitly casting the deserialized object to *Point*.

## Question 6

This question was attempted by 50 per cent of candidates.

### Part a

(i) The correct answer, given by 87 per cent of candidates, was that the output was `Sam`.

Incorrect answers were:

- `Jack`

  `Sam`
- `Sam`

  `Jack`
- `Jack`

(ii) The correct answer, given by 61 per cent of candidates, was that all exceptions are sub-classes of the *Exception* class. Incorrect answers given included 'runtime exceptions'; 'checked Exception'; and the most popular, 'Throwable'.

(iii) The correct answer, (A) was given by 83 per cent of candidates, with 13 per cent choosing (B) and the remainder choosing (C).

(iv) Most candidates (70 per cent) understood that `ArithmeticException` is unchecked, with 83 per cent knowing that `ArrayIndexOutOfBoundException` and `NullPointerException` are also unchecked.

Most candidates (91 per cent) understood that `FileNotFoundException` is a checked exception, with 74 per cent also understanding that so is `MalformedURLException`.

### Part b

(i) The correct missing word for statement (1), given by 65 per cent of candidates, was *BLOCKED*. Incorrect answers seen included *blocking*, *collision*, *sleeping*, *deadlock* and *running*.

The correct missing word from statement (2), *NEW*, was given by 30 per cent of candidates, with various wrong answers seen, such as *run*, *ran*, *runnable*, *deadlock* and *idle*.

(ii) The correct answer, given by 83 per cent of candidates, was (B), the 'thread deadlock' problem in thread programming is when thread *x* has the key that thread *y* needs in order to continue; and thread *y* has the key that thread *x* needs. Statements (C) and (D) were the wrong answers equally chosen by the remaining candidates.

(iii) Correct answers to the true or false questions posed were:

(A) TRUE [91 per cent answered correctly]

(B) FALSE [22 per cent answered correctly]

(C) TRUE [91 per cent answered correctly]

(D) FALSE [09 per cent answered correctly]

## Part c

52 per cent of candidates attempting this question achieved full credit. The remainder achieved partial credit, except for the 9 per cent who made no attempt.

Candidates were asked to add exception handling to the *SourceViewer* class, and instructed to include at least two *catch* blocks in their exception handling. Choice of the exceptions to handle was where most candidates lost credit.

To achieve full credit candidates had to handle the `MalformedURLException` thrown by the *URL* object. Additionally, the *read()* and *openStream()* methods could potentially throw an `IOException`. Therefore for full marks, catch blocks could have been one of the following combinations:

- `MalformedURLException; Exception.`
- `MalformedURLException; IOException.`
- `MalformedURLException; IOException; Exception` (Exception not needed but good practice to include in case anything has been overlooked).

Candidates lost marks by including only one catch block or by including catch blocks for exceptions that could not be thrown by the *SourceViewer* class, such as `FileNotFoundException` or `NumberFormatException`.

One mistake was seen with exception handling, putting the `MalformedURLException` catch block after the `IOException` catch block. The compiler visits catch blocks in order, and the `MalformedURLException` would be dealt with by the `IOException`, making the `MalformedURLException` catch block redundant, which the compiler would flag (`error: exception MalformedURLException has already been caught`).

A possible correct answer would be as follows (text that is part of the answer is indicated in bold):

```
import java.io.*;
import java.net.*;
public class SourceViewer1{
   public static void main(String args []){
      try {
         URL u = new URL(args [0]);
         //possible MalformedURLException above
         Reader r = new BufferedReader
         (new InputStreamReader(u.openStream()));
         //possible IOException (openStream())
         int c = -1;
         while ((c = r.read())!= -1) {
            System.out.print((char)c);
            //possible IOException (the read() method)
         }
      }
      catch (MalformedURLException e) {
         System.err.println(e);
      }
      catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

## Conclusion

The majority of candidates demonstrated a good grasp of Java, with a minority demonstrating an excellent understanding of the subject. A significant minority were not well prepared, and would benefit from more hands-on programming practice. The examiners also noted some answers that were ungrammatical and hard to comprehend, indicating that some candidates might have been able to improve their mark with a better standard of written English.