Examiners' commentaries 2015–16

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

General remarks

Introduction

Part A of the examination consisted of three questions on programming in Java, drawn from the topics covered in Volume 1 of the subject guide. Candidates were asked to answer two questions from this section. There were 25 marks available from each question, giving 50 marks in total from section A. Similarly, section B covered topics from Volume 2 of the subject guide, with candidates asked to choose two questions from three, and with 50 marks available in total.

The examination was done well on the whole, with an average mark for the paper of 67 per cent. The following report gives details on good answers as well as some common mistakes.

Comments on specific questions

Part A

Question 1

For part (a) almost all candidates correctly answered 11, showing understanding of how casting to an `int` works – everything after the whole number is cut off, with no rounding.

In part (b) the answer was often right, but some candidates thought that the answer would always be zero because `Math.random()`, in the following statement, would return zero:

```
int r = (int)Math.random()*2;
```

Math.random() produces a number between zero and one, exclusive (i.e. the number will never actually be zero or one). If the number is multiplied by two, giving a number between zero and two, and then cast to `int`, the result will be zero or one. Clearly this was the intention of the programmer, thus simulating the two possibilities of tossing a fair coin, heads or tails. However, the cast is done before the multiplication, meaning that zero is then multiplied by 2, so the result will always be zero. In order to get the desired result, brackets should have been used as follows:

```
int r = (int)(Math.random()*2);
```

Most candidates understood in part (c) that the *Laurel* class would compile. Not all candidates could give a complete explanation why the *Hardy* class would not compile, writing such things as because the *Hardy* constructor needs a parameter, or because the *Hardy* class is trying to make an object without a variable. A complete answer would have said that *Laurel* does not have a constructor, so the default empty constructor is supplied by the compiler and the attempt to make an object with no parameters succeeds. *Hardy* has overwritten the empty constructor by including an explicit constructor, hence class *Hardy* does not have a no-argument constructor and the attempt to make a no-parameter object fails.

Part (d)(i) was almost never answered completely correctly, with quite a number of candidates incorrectly stating that *this* was used to avoid ambiguity. Closer to the correct answer were those candidates who said the statement was setting the value of the *location* variable, and closer still were those candidates who said that the statement was calling the other *method* called *Jack*. Very few candidates gave a completely correct answer, i.e. stating that the statement, which was located in a no-argument constructor, was calling the other, one-argument constructor in the *Jack* class.

Part (d)(ii) was answered even more badly than the first part of the question. Very few candidates could even write a constructor that would compile. One answer often given was:

```
public Jill (String l, String t){
   super(l,t);
   //calling a two argument constructor in the parent
   class
}
```

Since there is no two-argument constructor in the parent class, the above would give a compilation error. Other candidates tried to use getters in their constructors to initialise the instance variables, or wrote constructors with no arguments, then tried to use the parameters that the constructors did not have. One student wrote that you cannot use *super* and *this* in a constructor, which was nearly right. You cannot use *super* to call a parent class constructor, together with *this* to call a constructor from the same class, but you can use *super* to call a parent class constructor, and *this* to avoid ambiguity as follows:

```
public Jill (String location, String thing){
   super(location);
   this.thing = thing;
}
```

Part (e) was answered correctly more often than part (d). For full credit, candidates needed to be convincing; there were no marks for writing that the method would throw an exception as it was trying to access an array element that did not exist, or because it was searching past the end of the array. Candidates got some marks for saying that the method tried to access array element num[-1], which did not exist, thus causing the exception. For full credit, candidates needed to explain why the method would try to access an element that did not exist, and explain which statement would cause the error. A good answer could have said that the method will search from the end of the array to the beginning, decrementing *i* by 1 at each iteration. When the search item is found in the array, the final decrement does not happen, and the value of *i* will be zero, or greater than zero, when the *for* loop ends. When the item is not found, the final decrement happens, setting *i* to -1, and causing the `ArrayIndexOutOfBoundsException` when the method reaches the `return (a[i]==find);` statement.

Although not at all necessary for full credit, the best answers discussed the problem and proposed a solution, for example:

```
public static boolean linearsearch2(int[ ] a, int find){
   for (int i=a.length-1; i>=0; i--)
      if (a[i]==find) return true;
   return false;
}
```

The average mark for this question was lower than for every other question except for Question 6. Poor answers to part (d) contributed significantly to this low mark. This was the most popular question, with 94 per cent of candidates attempting it.

Question 2

In part (a) some candidates wrote that it is not possible to make an object/instance of the class. While correct, a complete answer would say that the compilation error would happen because an abstract class cannot be instantiated. Some candidates wrote that an abstract class cannot be 'initialised'.

In part (b) some candidates gave the wrong answer as they failed to appreciate that, unlike an abstract class, an interface *must* have all abstract methods.

Part (c)(i) many candidates only wrote:

```
class Cat implements Animal{}
```

which was not enough for full credit.

A complete answer would implement the *numberOfFeet()* method:

```
class Cat implements Animal{
   //private int feet;
   public int numberOfFeet(){
      return 4;
      //return feet;
   }
}
```

Full credit was only given when the *numberOfFeet()* method was explicitly *public*; all interface methods are implicitly *public* so missing out the access modifier in the implementing class would give a compilation error.

A good answer to (c)(ii) was not often seen; many answers were very poor and did not get full credit. Some candidates failed to understand that their answer needed to extend the *Cat* class; many could not or did not write a constructor. Despite this, a very simple answer, as follows, would have achieved full credit:

```
class PetCat extends Cat{
   String name;
   PetCat(String s){
      name=s;
   }
   String getName(){
      return name;
   }
}
```

Part (d) was usually right. Most candidates understood that statement 2 would have produced the exception, although a minority of candidates thought that statements 1 or 5 would give the error. Similarly, in part (e)(i) most candidates understood that *speak()* had been overridden in *Robot*, although some candidates confused this with overloading and wrote that the constructors in *Machine* had been overridden. Or more accurately, some candidates wrote that the *methods* in *Machine* had been overridden, reflecting that many candidates had trouble both writing and recognising constructors. Pleasingly, most candidates gave the correct answer for (e)(ii):

```
The machines are rising
I am your robot overlord
the robot uprising has begun
```

92 per cent of candidates attempted this question.

Question 3

Part (a) asked candidates to say whether five statements about graphical programming were true or false. All candidates understood that statement A was true, most understood that B was true, most thought that C was true when it was in fact false, the majority correctly thought that D was true, and the consensus was that E was false, even though it was actually true.

Part (b) was answered correctly on the whole; most candidates wrote two inner classes, each implementing the `ActionListener` interface, and successfully added one to each button. A small minority lost marks by writing one inner class and querying the event source, and a small number of candidates wrote complicated inner classes, when they could have been very simply written as follows:

```
class button2Listener implements ActionListener{
   public void actionPerformed (ActionEvent e){
      button1.setText("CLICK ME NOW!");
      frame.repaint();
   }
}
class button1Listener implements ActionListener{
   public void actionPerformed (ActionEvent e){
      button2.setText("Don't click him again!");
      frame.repaint();
   }
}
```

Part (c)(i) and (ii) were either done very well, or very badly. Some candidates put *repaint()* into a loop in their *paintComponent()* method, failing to realise that this effectively gives a loop inside a loop. The repaint manager will try to collapse all the requests into a single one, and since it will be getting many requests, this would be likely to seriously slow down or stop the animation. Another common mistake was forgetting to slow down the animation using `Thread.sleep` or otherwise. Some candidates also used `frame.getHeight()` in their calculations to keep the circle visible, when `drawPanel.getHeight()` would have given more accurate results. Some candidates used 500 or another number in their calculations; using *getHeight()* to find the exact height of the draw panel would have given a better result, and a more scalable one.

This was not a popular question, attempted by only 11 per cent of candidates, but it had the highest average mark of all six questions.

Part B

Question 4

In answering part (a) about half of all candidates thought that it was true that child classes could only be serialized if their superclasses were serializable, even though it was false, while the majority of candidates understood that static variables could not be serialized and that transient variables do not have their state saved on serialization.

Part (b)(i) was answered fairly well, but many candidates did not give complete answers. Candidates wrote things such as 'it will prevent another class from accessing the class' or 'the constructor would only be available to the class' or similar answers that applied the definition of the private access level in Java. Partial credit was given for answers that said such things as 'so it cannot be used by another class', or 'because it is a utility class'. A complete answer would have said that the constructor was private to prevent an instance of the class being made, since it does not make sense to instantiate a class that only contains static methods.

Most candidates got all marks for (b)(ii), understanding the `String.format()` method well enough to know that (B) was the correct answer. Similarly, most candidates had no trouble answering (b)(iii) correctly, knowing that an `ObjectOutputStream` would be needed for deserialization purposes.

Part (b)(iv) was answered correctly by about half of those that attempted it; many did not give an answer. A common error was forgetting to return an `ArrayList`, despite the method being typed as an `ArrayList` method. A minority of answers simply used their stream to read in an object, without attempting the `ArrayList` cast. A more common error was forgetting to throw or handle exceptions. Different answers were possible, but a good answer would be:

```
public static ArrayList<SL> fromSerialized(String filename) {
   ArrayList<SL> teams = null;
   ObjectInputStream in;
   try {
      in = new ObjectInputStream(new FileInputStream(filename));
      teams = (ArrayList<SL>) in.readObject();
      in.close();
   }
   catch (FileNotFoundException e) {
      System.err.format("File not found! %s", e);
      return null;
   }
   catch (Exception e) {
      System.err.format("Error reading from file: %s", e);
      return null;
   }
   return teams;
}
```

In part (c), 47 per cent of candidates got all three answers correct, 42 per cent got one answer correct, and 11 per cent gave all incorrect answers. The answers were almost twice as good as those that would have been given if all candidates had picked one of the three answers at random, since 58 per cent of candidates correctly thought that *Showfile2* would work, 58 per cent of candidates correctly thought that *ShowFile3* could enter an infinite loop, and 58 per cent correctly thought that *ShowFile4* could end with a `FileNotFoundException`.

Candidates did not have to explain why the programs worked as they did for full credit, but were expected to notice that *ShowFile3* had a do/while loop that would continue while a `boolean` variable was false. Inside the loop the variable would become true if the file entered by the user existed.

However, if the file did not exist, the variable stayed false, since no statements had been written to ask the user to enter another file and to read in the `String` entered, meaning that the loop repeatedly checked to see if the first (and only) file name entered by the user belonged to an existing file.

Candidates were expected to notice that *ShowFile4* entered a loop if the file with the name given by the user, called *file* by the program, did not exist. It asked the user for another file name and checked to see if a file with that name existed; the program called this variable *file2*. Once the loop ended the class then tried to access the *file* variable, hence causing the `FileNotFoundException`.

72 per cent of candidates attempted this question.

Question 5

Most candidates answered part (a)(i) correctly, understanding that *static* final variables must be initialised, and if they are not, the compiler will flag this as an error. Oddly, a common error seen from a minority of candidates was to write that the problem was the *s* variable had not been initialised. Candidates who wrote this showed that they did not understand that instance variables do not normally have to be initialised.

Most candidates understood that the capital letters indicated that the *FORINSTANCE* variable was static and final, i.e. a constant, but for full credit needed to make it clear that they understood that this was a convention, and was not something that the compiler would enforce.

In part (b) candidates needed to explain that *askToDance()* is a final method, and could not therefore be overridden by the *Happier* class, and in part (c) that since *area* was a final variable, the *calcArea()* method would not be able to give it a new value. Almost all candidates gave correct answers to these two questions.

In part (d)(i) three candidates thought that the answer was (A): the class would not compile because the bracket closing the class was missing. And they were correct because a mistake in the paper meant that the bracket had been accidentally deleted! Most candidates did not notice the error and gave the answer intended by the examiner which was (C). Giving this answer showed that candidates understood that the program would compile, would throw an exception, and that once the exception was thrown the program would stop. Answering (A) showed that candidates were paying more attention to detail than the setter and moderators were. Both answers were marked as correct. One candidate wrote that the class closing bracket was missing, noted that s/he assumed that this was a mistake and gave their final answer as C; this candidate deserved a bonus mark for being right twice over!

A minority gave their answer as (D), thus showing that they did not understand that once an exception is thrown, a program stops.

A minority of candidates thought that the program would not compile because the exception had not been thrown or handled; these candidates had not noticed that `NewException` was a child class of `RunTimeException`, and hence was not an exception checked by the compiler.

A variety of different answers were seen for (d)(ii). Usually answers were complicated as well as wrong. The examiners were expecting candidates to realise that it was a simple matter of adding either a literal or a variable `String` to the *onlyPositive()* method as in the following answer:

```
static int onlyPositive (int x) throws NewException{

   if (x<0) throw new NewException(x +” is a negative number,
   positive numbers only”);

return x;

}
```

Doing this would call the constructor in the *NewException* class that takes a `String` parameter, would have returned the correct message when the exception was thrown, and would have gained full marks.

Even when candidates managed to give an answer similar to that above (adding a literal or variable `String` to the statement), some candidates did not seem to appreciate how to display the contents of the x variable to the user, writing statements such as:

```
if (x<0) throw new NewException("<value of x> is a negative
number, positive numbers only");
```

and losing some credit.

Many candidates added *try/catch* blocks to their main method, introducing a variety of additional errors. Some used the *x* variable in their catch block messages, seeming to forget that *x* was not in scope in the main method, so attempting to access it would give a compilation error. Others got around this problem by adding `int x = -2;` to their main methods. Some simply wrote in their catch block "`-2 is a negative number…`" Others used the *z* variable for the error message; since *z* would be zero if the program entered the catch block, the user would see the message:

```
0 is a negative number, positive numbers only
```

Another error seen several times was that candidates added to the *onlyPositive()* method a check on the value of *x* such that, if *x* was less than zero, the message was printed to standard output. Others had the message returned if x was less than zero; since the method was an `int` method, returning a `String` would cause a compilation error. One candidate attempted to get around this by making the *onlyPositive()* method a `String` method; however, this would have caused a compilation error in their main method, since in the statement `z = onlyPositive(-2);` *z* is an `int` variable.

The best answers seen added some `String` formatting to the message; not necessary for full credit, but a nice touch. The minority of candidates who answered this question correctly are to be congratulated.

86 per cent of candidates attempted this question.

Question 6

In parts (a) and (b) candidates were asked to choose which one of three options was true. If part (a) was an election, then statement (A) would have been voted true by 50 per cent of the electorate, statement (B) by 44 per cent and (C) by 6 per cent. This meant that the one true answer (the *Runnable* interface gives a job to a *Thread* object) was narrowly voted the winner. In part (b) no-one was fooled by statement (A), 78 per cent voted for the correct answer, statement (B), but 11 per cent of the electorate voted for statement (C) and 11 per cent for (D).

For part (c)(i) most candidates understood that the method was 'listening' to the machine with the IP address 190.165.1.103. A few candidates gave answers that were not specific enough, such as the method was listening to a router, or to a server. In (ii) the majority understood that the method printed any data that it 'heard' to the screen (or to standard output), while a few candidates wrote such things as the method 'gets a stream', 'reads a character' or 'reads data'.

In part (d) 61 per cent of candidates chose (A), showing understanding that the `Thread.sleep(100);` statement in the *doMoreStuff()* method was a way of forcing control to move back to the *main* thread. The rest of the candidates were evenly split between (B) and (C), with some of the candidates who chose (C) explaining that thread scheduling is unpredictable.

In part (e) 94 per cent of candidates understood that the *ThreadedServer* assigned a thread to each new connection; 60 per cent understood that statement (B) was false; 73 per cent understood that after accepting a connection to a client, the program would print the client address to the screen; and 67 per cent of candidates understood that statement (D) was false.

In part (f) most candidates did not understand that, because of the following statements in the *main* method, there were two threads trying to gain access to resources of the same object:

```
P it= new P();

T1 t1 = new T1(it);

T2 t2 = new T2(it);
```

Thread *t1* would start the *g()* method, and thread *t2* would start the *f()* method. Since both these methods were synchronized, this would mean that once one method started, it could not be interrupted by the other thread until it concluded. As both synchronized methods contained an infinite loop, once one thread got control, the other thread was effectively locked out. This meant that the user would see either 'hello' followed by a new line printed on the screen repeatedly or the user would see an infinite loop of 'goodbye', with each 'goodbye' on a new line. That is, the user would see all hellos, or all goodbyes.

Candidates only needed to give the correct output for full marks, which a tiny minority of candidates did. Partial credit was given to candidates who wrote that the user would see an infinite loop of some sort, and partial credit was also given to the candidate who wrote that there was conflict because both threads wanted access to the resources of the same object, and that as a result there would be no output.

Part (f) was the question found most challenging by candidates on the entire paper; very few gained full credit, and quite a number could not even correctly identify that the program would give an infinite loop when run. This contributed to Question 6 having the lowest average mark for the paper. Less than half of candidates, 44 per cent, attempted this question.

Examiners’ commentaries 2015–16

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

General remarks

Introduction

Part A of the examination consisted of three questions on programming in Java, drawn from the topics covered in Volume 1 of the subject guide. Candidates were asked to answer two questions from this section. There were 25 marks available from each question, giving 50 marks in total from section A. Similarly, section B covered topics from Volume 2 of the subject guide, with candidates asked to choose two questions from three, and with 50 marks available in total.

The examination was done well on the whole, with an average mark for the paper of 66 per cent. The following report gives details on good answers as well as some common mistakes.

Comments on specific questions

Part A

Question 1

For part (a) almost all candidates correctly answered 5, showing understanding of how casting to an `int` works – everything after the whole number is cut off, with no rounding.

In part (b) some candidates thought the output would be 500, whereas others thought it would be zero. Some of those that answered zero thought that *Math.random()* returned zero; they did not notice the problem with the cast. In fact, *Math.random()* produces a number between zero and one. The *CoinToss1()* method casts the output to an *int*. The result of the cast will be always be zero because of the way that casting works. Zero is then multiplied by two, but zero times anything at all is still zero. In order to get the result that was probably desired by the author, brackets should have been used to force the multiplication to be done before the cast as follows:

```
int r = (int)(Math.random()*2);
```

Most candidates understood in part (c)(i) that the *Ant* class would compile. In (ii) candidates gave the reason why *Dec* would not compile, with many writing such things as ‘the attempt to make an object in main will fail because there is no parameter’, or ‘because *Dec* does not have a no-argument constructor’, although many candidates called the constructor a method, i.e. ‘the method *Dec* needs a parameter’. These and similar answers showed understanding, but a complete answer would have said that *Ant* does not have a constructor, so the default empty constructor is supplied by the compiler and the attempt to make an object with no parameters succeeds. *Dec* has overwritten the empty constructor by including an explicit constructor, hence class *Dec* does not have a no-argument constructor and the attempt to make an object with no parameters fails.

A few candidates wrote that *Dec* would not compile as the main method could not access the non-static constructor, or that *Dec* could not access the `String` variable *s*, because it was private; this showed a lack of understanding of the basics of Java programming.

Part (d)(i) was almost never answered completely correctly. Most students got nothing for this, writing things like 'it is inheriting the value of *location* from the parent class', or 'it is setting the value of *location*' or 'the purpose of the statement `super(l);` is to call the parent class of *Jill* with reference to the variable *l*.' It was as if candidates had been asked to answer the question without using the word 'constructor', but most likely it was because most candidates did not appreciate that *super* was being used to invoke the constructor of the parent class. A good answer would simply have said that the statement is calling the constructor in the parent class *Jack*.

A common mistake was to write that the purpose of the statement was to *inherit* the variable location from the class *Jack*, but inheriting is done by extending a class, so this answer indicates a very limited understanding of the basic underpinnings of Java programming.

Part (d)(ii) was answered even more badly than the first part of the question, with very few correct answers seen. The most common answer given was:

```
public Jack(){this.location = "down the hill";}
```

Other answers seen were:

```
Jack(){Jack("down the hill");}
```

and

```
public void setLocation(){Jack("down the hill");}
```

and

```
public Jack(String location){
   if location.equals("") this.location="down the hill";
   else this.location=location;
}
```

None of these answers gained any credit.

A minority of students gained some credit for writing an almost correct answer, in that their constructor had a parameter (*location*), although it should not have done since this would cause a compilation error, as the existing constructor had the same signature.

An even smaller minority gave the correct answer:

```
public Jack(){
   this("down the hill");
   /*'this' calls the other constructor in class Jack, with
   "down the hill" as the parameter*/
}
```

Part (e) was answered correctly more often than part (d). For full credit, candidates needed to be convincing; there were no marks for writing that the method would throw an exception as it was trying to access an array element that did not exist, or because 800 did not exist in the array, although some credit was given for writing that the method tried to access array element, *num[7]*, which did not exist thus causing the exception. For full marks, candidates needed to explain why the method would try to access an element that did not exist, and explain which statement would cause the error.

A good answer could have said that the method works as it should if the number searched for is in the array, i.e. it returns true. When searching for a number not in the array, the method would throw an `ArrayIndexOutOfBoundsException` because the method would search from the beginning of the array to the end, incrementing *i* by 1 at each iteration. When the search item was found in the array, the final value of *i* would be 6 or less. When the item was not found, the final increment would happen, setting *i* to 7, and causing the `ArrayIndexOutOfBoundsException` when the method reached the `return (a[i]==find);` statement.

The average mark for this question was lower than for every other question except for Question 6, and lower than the average mark for all questions. Poor answers to part (d) contributed significantly to this low mark. This was the most popular question, attempted by 98 per cent of candidates.

Question 2

In part (a) a complete answer would say that an abstract class cannot be instantiated (note that 'initialised' means giving something a starting state), and not just that it is not possible to make an object of the class.

In part (b) some candidates gave the wrong answer (C: the class would compile correctly but there would be a run-time error in any extending classes because of the non-abstract method) since they failed to appreciate that an abstract class can have concrete methods.

Part (c)(i) many candidates only wrote:

```
class Dog implements Animal{}
```

which was not enough for full credit.

A complete answer would include an implementation of the *isCarnivore()* method:

```
class Dog implements Animal{
   public boolean isCarnivore(){
      return true;
   }
}
```

Some candidates made the mistake of writing 'extends' instead of 'implements' in the above. Other common mistakes were adding the modifier 'static' to the *isCarnivore()* method, and giving the method a parameter that was not an instance variable of the class and so would cause a compilation error. Full credit was only given when the *isCarnivore()* method was explicitly *public*; all interface methods are implicitly *public* so their implementation needs to be explicitly public.

Many correct answers to (c)(ii) were seen. A few candidates failed to understand that their answer needed to extend the *Dog* class, and a minority could not write a correct constructor, or did not write a constructor at all. Most gave a correct answer similar to the following:

```
class PetDog extends Dog{
   int age;
   PetDog(int a){
      age=a;
   }
   int getAge(){
      return age;
   }
}
```

Part (d) was usually right, almost all candidates understood that statement 5 would have produced the exception.

In part (e)(i) the constructors in the *Machine* class were overloaded. While many candidates understood this, many called the constructors 'methods', or simply wrote that overloading happened in *Machine*, without mentioning the constructors. A few confused overloading with overriding, and said that the *speak()* method was overloaded in class *Robot*.

Most candidates gave the correct answer for (e)(ii), although a minority thought that the first and second line of output would be the same. The actual answer was:

```
The machines are rising
I am your robot overlord
the robot uprising has begun
```

91 per cent of candidates attempted this question.

Question 3

Part (a) asked candidates to say whether five statements about graphical programming were true or false. All candidates understood that in order to handle events a GUI must implement a listener interface (statement A); and 80 per cent understood that an inner class implementing a listener interface for each `JButton` is the accepted solution to listening to more than one button (statement B). All candidates thought that statement C was true when it (1) was false and (2) contradicted statement B. A 60 per cent majority correctly thought that D was false, and an 80 per cent correctly identified statement E as false.

Part (b) was answered correctly on the whole, with a few candidates losing credit by writing one inner class and querying the event source; otherwise, correct solutions were seen.

Part (c)(i) and (ii) were done well by most candidates. Some minor mistakes were seen, including forgetting to slow down the animation using `Thread.sleep` or otherwise, and using *frame.repaint()* in the *paintComponent()* method, when using *repaint()* on the draw panel only would have been better. Additionally, most candidates used a number (usually 500) in their calculations to keep the circle visible, when *drawPanel.getWidth()* would have given more accurate results, and more scalable ones. Below is one possible solution:

```
class CircleDrawPanel extends JPanel{
   public void paintComponent (Graphics g){
      super.paintComponent(g);
      Graphics2D g2=(Graphics2D)g;
      g2.setColor(color);
      g2.fillOval(x,y,diameter,diameter);
      x++;
      repaint();
      try{
         Thread.sleep(10);
      }
      catch(Exception e){}
      if (x == drawPanel.getWidth()) x = 0;
   }
}
```

Note that the statements in bold are the answer to part (i), and the statement

```
if (x == drawPanel.getWidth()) x = 0;
```

is the answer to part (ii). Other correct answers were possible, such as

```
if (x == drawPanel.getWidth() - diameter) x = 0;
```

or

```
if (x > 500) x = 0;
```

In general, any answer that would restart the animation from the starting position once the circle had left the frame/draw panel, or before it left the frame/draw panel, was accepted by the examiners.

This was not a popular question, attempted by only 11 per cent of candidates, but it had the highest average mark of all six questions.

Part B

Question 4

In answering part (a) about only 91 per cent of candidates knew that it was false that all classes are serializable by default; 77 per cent knew that it was true that static variables cannot be serialized; and 74 per cent understood that the third and final statement was false.

Part (b) was intended to ask candidates to 'Consider the *SL* and *SLUtils* classes below:'. Unfortunately, what was actually printed on the paper was the *SLTest* and *SLUtils* classes. Candidates still had all the information that they needed in order to answer the question, and indeed most successfully did. Comparing the average mark of 19 to the average mark of 18 for the similar question in the zone A paper, indicates that this mistake (which was not repeated in the other zone) did not disadvantage candidates. Additionally, 82 per cent of candidates attempted the question, compared to 72 per cent attempting the similar question in the zone A paper.

Part (b)(i) was answered fairly well, but many candidates gave slightly vague answers that hinted at understanding such as 'It cannot be used by another class'; 'It will prevent another class from accessing'; 'Only SLUtils can access its constructor'; 'It is a utility class'. A good answer would have said that the effect of making the constructor private would be to prevent an instance of the class being made. Some candidates wrote that it would prevent the class from being 'initialised' – the examiners gave benefit of the doubt, assuming that candidates meant 'instantiated'.

Most candidates got all marks for (b)(ii), understanding the `String.format()` method well enough to know that (C) was the correct answer, with candidates using the *SLTest* class to answer the question; 82 per cent answered correctly, compared to 75 per cent for the similar question in zone A.

Similarly, 91 per cent of candidates answered (b)(iii) correctly, knowing that an `ObjectInputStream` would be needed for deserialization purposes.

Part (b)(iv) was answered badly by many. The most serious mistake made was making a local `ArrayList` with the same name as the parameter (teams). This would mean that an empty `ArrayList` would be written to the file, so that the method would fail in its objective. Less serious mistakes were forgetting to write to the file, forgetting to close the file, or forgetting to handle exceptions.

```
public static void serializeToDisk(ArrayList<SL> teams) {
   FileOutputStream fos;
   ObjectOutputStream oos;
   try {
      oos = new ObjectOutputStream(new FileOutputStream("teams.ser"));
      oos.writeObject(teams);
      oos.close();
   }
   catch (FileNotFoundException e) {
      System.err.format("File not found! %s", e);
   }
   catch (Exception e) {
      System.err.format("Error reading from file: %s", e);
   }
}
```

In part (c), 61 per cent of candidates got all three answers correct and 3 per cent gave all incorrect answers. The remaining 36 per cent gave one right answer only. Looking at the answers regarding each individual file, 69 per cent of candidates correctly thought that *Showfile2* would work, 89 per cent of candidates correctly thought that *ShowFile5* would not compile, and 61 per cent correctly thought that *ShowFile6* could enter an infinite loop.

Candidates did not have to explain why the programs worked as they did for full credit, but were expected to notice that in *ShowFile5* `System.in` and `System.out` had been confused, causing the compilation errors. In *Showfile6* the do/while loop tried to make a file with the name given by `args[0]`. If the file existed, a `boolean` variable was made true, causing the loop to end. If the file did not exist, the user was asked for another file name, and the boolean variable remained false, so that the loop would continue. On the next iteration of the loop the program again tried to make a file from `args[0]`, rather than from the new file name supplied by the user. Since this had already failed, it would fail again, and the loop would be infinite.

Question 5

Many candidates answered part (a)(i) correctly, understanding that *static final* variables must be initialised, and if they are not, the compiler will flag this as an error. A number of wrong answers were seen, such as that the statement in the *Zx* constructor should be `n = name`; or that it should be `this.name = n`; or that a variable cannot be both static and final; or, the worst answer seen, that the *Zx* constructor did not have a type. Another answer was that the *name* variable had not been initialised as it should have been since non-primitive variables have no default value (which they do, *null*). All of these answers showed a failure to grasp the basics of Java programming.

In (a)(ii) most candidates understood that the capital letters indicated that the *EXAMPLE* variable was static and final, i.e. a constant, but for full credit candidates needed to make it clear that they understood that this was a convention, i.e. writing something along the lines of 'because EXAMPLE cannot be changed' would not achieve full credit.

In part (b) most candidates explained that the error the compiler would find was that the variable *circumference* was final, and thus could not be given a value by the *calcCircumference()* method.

In part (c) most candidates wrote that since class *Good* was final it could not be extended by class *Better*, and the compiler would flag this as an error. A few gained partial credit by writing that the error was that class *Good* had a method being overwritten in the child class *Better*, but a method in a final class could not be overridden. Some answers were seen that showed a lack of understanding of Java basics, such as that the statement `super(n);` was illegally trying to access a private variable; that the statement was the problem for some other reason; or that the constructor in *Good* would not be able to access the *name* variable because it was private.

In part (d)(i) some candidates thought that the answer was (A): the class would not compile because the exception had not been thrown or handled; these candidates had not noticed that `NewException` was a child class of `RunTimeException`, and hence was not an exception checked by the compiler. A large minority answered (D), showing that they thought a program would continue after an exception had been thrown. The majority answered (C), showing that these candidates understood that the program would compile, would throw an exception, and that once the exception was thrown the program would stop. The best answers explained that the program would compile because the exception was a `RunTimeException`; however, this was not necessary for full credit.

In part (d)(ii) most answers were wrong. In fact, almost all were, but in very varied and inventive ways, including: try/catch blocks in the main method that attempted to access an out of scope variable; try/catch blocks in the *onlyPositive()* method; returning the statement as a `String` from the `int` *onlyPositive()* method; adding a *toString()* method to the *AbsVal* class; adding a *NewException* constructor that would print the message `x + "is a positive number…"`, despite *x* being out of scope in the *NewException* class; and adding to the *onlyPositive()* method a statement such as `if (x < 0) System.out.println(//the message);`. The final answer in particular ignored that the question asked that candidates change the *AbsVal* class so that if the exception was thrown then the user would see the message; writing the message with `System.out.println()` if *x* was less than zero had nothing to do with the exception being thrown.

The examiners were expecting candidates to realise that all they had to do was to add a `String` to the *onlyPositive()* method as in the following answer:

```
static int onlyPositive (int x) throws NewException{
   if (x<0) throw new NewException(x +" is a negative
   number, positive numbers only");
return x;
}
```

Doing this would call the constructor in the *NewException* class that takes a `String` parameter, would have returned the correct message when the exception was thrown, and would have gained candidates full marks.

The best answers seen added some `String` formatting to the message; not necessary for full credit, but a nice touch. The minority of candidates who answered this question correctly are to be congratulated.

84 per cent of candidates attempted this question.

Question 6

In part (a) the three states of a live thread (runnable, running and blocked) were variously described as: start, run, runnable; start(), sleep(), block(); establish, ready, run; sleep(), wait(), notify(); new, running, runnable; wake up, sleep, blocked. *Runnable* and *blocked* were popular choices; *running* less so. The majority of candidates did not get full credit for this question.

In part (b) most candidates understood that the lost update problem was when a thread 'wakes up' and continues operating on a value it had read before going to sleep, not knowing that another thread has changed it. However, 42 per cent of candidates thought that the lost update problem was 'when a thread collision happens and the thread loses the data on the top of its call stack'; since this answer is nonsense, a lot of candidates were clearly guessing.

For part (c) most candidates understood that the method was 'listening' to the machine with the IP address 190.165.1.103, and that it was listening to port 4242.

In part (d) 8 per cent of candidates chose (A), 42 per cent chose (B) and 50 per cent chose (C). This means that only 42 per cent of candidates attempting the question understood that the `Thread.sleep(100);` statement in the main method would give control to the t thread, making (B) the most likely output.

In part (e) 87 per cent of candidates understood that the *ThreadedServer* assigned a thread to each new connection; only 43 per cent understood that statement (B) was false; 64 per cent understood that after accepting a connection to a client, the program would print the client address to the screen; and 73 per cent of candidates understood that statement (D) was false.

In part (f) a popular answer was to say that the user would see either `hello` or `goodbye`. Sadly this answer was very far from the truth. In fact there were two threads, *t1* and *t2*, each attempting to access the resources of the same object. Once one thread got control, it would keep control, as the method it would run (1) had an infinite loop, and (2) was synchronized so the thread could not be interrupted while the method was running . One thread's method would repeatedly print hello, and the other goodbye. This meant that the user would see an infinite loop of 'hello' or an infinite loop of 'goodbye', each printed on a new line.

Candidates only needed to give the correct output for full marks, which a tiny minority of candidates did. Partial credit was given to candidates who wrote that the user would see an infinite loop of some sort.

Part (f) was the question found most challenging by candidates on the entire paper; very few gained full credit, and quite a number could not even correctly identify that the program would give an infinite loop when run. This contributed to Question 6 having the lowest average mark for the paper. About one in three candidates, 34 per cent, attempted this question.