# Examiners' commentary 2018–2019

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

## Comments on specific questions

### Question 1

This question was attempted by almost all candidates.

**a. Answer**

i.
```
Dog name is null
Dog age is 0
```

ii. (B) When a class does not have a constructor, the JVM automatically adds a default, no-argument constructor.

**Comments**

In answering part (i) a few candidates lost some credit by writing only 'null' and '0'; however most candidates demonstrated their understanding that uninitialised instance variables have default values. A small number did not understand this and gave the output as:

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

Part (ii) was always answered correctly.

**b. Answer**

```
636 House 6
636 hello Woo
42 hello Woo
1000 The impossibility of now
```

**Comments**

This question was one of the worst attempted on the paper, with only about a third of candidates giving completely correct answers. Mistakes with the final line of output were the single biggest error with this question, with less than half of all candidates who attempted the question giving the correct output.

This question tested students' understanding of constructors and inheritance. Hence students were expected to understand that in the *SonOfWoo* constructors, the keyword *super* was being used to call the superclass constructor, and that since the superclass had three constructors, the one called depended on the parameters given with *super*. For example `super(int, String)` would call the Woo constructor with the parameter list `int, String.` Many, although not all, candidates demonstrated their understanding of this with their answers; in fact some candidates gave the first three lines of output completely correctly, but stumbled with the final line of output.

The no-argument constructor in *SonOfWoo*, which has no statements in its body, makes the *sow4* object. This caused many students to write that there would be no output, or to give `0 null` (or just `null`) as their answer.

This question was testing whether or not students understood that if there was not an explicit call to the superclass constructor using *super* then there would be an implicit call. The compiler would insert `super();` hence the implicit call is to the superclass's no-argument constructor. Therefore, the no-argument constructor in *Woo* would be called, and this would give the *sow4* object's fields the default values: `1000` and `"The impossibility of now"`.

Other errors were to do with the order of printing of the objects' fields. The output of the *SonOfWoo* class is given by printing four *SonOfWoo* objects. *SonOfWoo* does not have a *toString()* method, while *Woo*, the direct parent class of *SonOfWoo*, has a *toString()* method. If the JVM does not find a *toString()* method in the source code of the object it is printing, the JVM will look in the inheritance chain, and use the first *toString()* method that it finds when going up the inheritance chain. Since the *Woo* class is the direct parent class of *SonOfWoo*, this will be the first *toString()* method that the JVM will find when it looks up the inheritance chain. Hence *SonOfWoo* objects will be printed using the *toString()* method from *Woo*.

The *toString()* method in the *Woo* class specifies that the fields of the object will be printed on one line with a space between them in the order `int`, `String`.

The following mistakes relate to the *toString()* method:

- Giving output in the order `String`, `int`
- Giving some output in the correct order (`int`, `String`) and some output in the order `String`, `int`
- Giving the output over two lines, rather than one (for eight lines of output in total).
- Giving the output in the wrong order (`String`, `int`) and over two lines rather than one.

The second mistake was particularly puzzling, in that these candidates surely could not have thought that the *toString()* method was changing as the output was being printed. However, it is likely that so many mistakes were made because many candidates simply did not realise that the *toString()* method in *Woo* was being used to print the fields of *SonOfWoo* objects, which is a fairly basic error.

**c. Answer**

i.

```
public void speak(){
   System.out.println("I am your robot overlord");
}
```

ii.

```
public void broadcast (){
   System.out.println("the robot uprising has begun");
}
```

**Comments**

Part (i) was answered well with 90 per cent of candidates demonstrating an understanding of overriding and gaining full credit. The remaining candidates wrote a *speak()* method with a String parameter, and gained no credit as methods written to override a superclass method must have the same parameter list (in this case an empty list) as the method they are overriding. Therefore any parameter for the *speak()* method was an error.

Part (ii) was not answered quite as well, with two mistakes seen:

- About 20 per cent of candidates gained no credit for this question because they wrote a method with a String parameter. A *broadcast()* method with a String parameter would be flagged by the compiler with: error: method broadcast(String) is already defined in class Robot. Methods are overloaded by giving methods with the same name a different parameter list. In this way constructors, as seen in part (b), are overloaded; a class can have an arbitrary number of constructors, provided that their parameter lists are different.
- Another 10 per cent of candidates gave their method parameters, usually a String and an int parameter. While this is overloading, and would cause no problems for the compiler, these candidates lost some credit. The parameters were quite pointless as they were not needed or used in the method, and would just make invoking the method more difficult for users who would have to think up arbitrary values for them.

## Question 2

This question was attempted by about 75 per cent of candidates.

**a. Answer**

i. (A) An abstract class can be instantiated. **FALSE**

(B) A class can implement any number of interfaces, but can only extend one abstract class. **TRUE/FALSE (see comment)**

(C) An abstract class cannot have concrete (i.e. non-abstract) methods. **FALSE**

(D) An abstract class must have the abstract key word in its class declaration. **TRUE**

ii.

```
public class ZedStuff extends Zed{
   public boolean lessThanZero(int x, int y){
      int i = x+y;
      return(i<0);
   }
}//other correct answers are possible
```

**Comments**

In part (i) the examiners decided to accept *true* or *false* as the correct answer to (B). This was because, on reflection, the examiners felt that this question should not have included the word 'abstract', since a class can only extend one class, whether or not it is abstract. This unclear wording implied that a class could always extend one abstract class, irrespective of whether or not the class extended any concrete classes. Overall, the question was answered correctly by most candidates, with only a few incorrect answers given to (C).

Part (ii) was usually answered correctly, although a minority of candidates copied the parameters of the *lessThanZero()* method to local variables and then used these local variables in determining what the method should return. No deduction was made, but candidates ought to have known that this was unnecessary.

**b. Answer**

i. `/*6*/ s = a.get(0);`

ii. `/*1*/ answers.size()`

`/*2*/ answers.get(index);`

iii. (A) `Arraylists` can be parameterised to object variables, and also to primitive variables. **FALSE**

(B) `Arrays` are much slower than `ArrayLists` because they are not directly mapped to memory. **FALSE**

(C) The enhanced for loop can be used for iterating through what Java calls collections. It can be used with an `ArrayList` because Java considers the class to be a part of the collections framework. **TRUE**

**Comments**

Very few candidates gave the correct answer to part (i). Many candidates answered that line 4 was the problem, while others answered that either lines 4 and 6 would both give a compilation error, or that lines 4, 5 and 6 would all give compilation errors. These candidates thought that an `ArrayList` could not be declared without a parameter, hence they considered the problem either was line 4, or started with line 4.

As Volume 1 of the subject guide discusses in section 7.3, `ArrayLists` do not have to be parameterised. The guide states:

> Prior to Java 5, any object could be inserted into a particular `ArrayList`; retrieved objects had to be cast back again to the correct type.

Hence the single problem is on line 6, because the statement tries to retrieve an object from the `ArrayList` without casting. The compiler flags this line with: `error: incompatible types: Object cannot be converted to String`

A correct statement would be: `s = (String)a.get(0);`

In part (ii) all candidates gained at least some credit for this question, since mostly correct answers were seen, with a few common errors. One mistake was to put a number in place of fragment 1, when `answers.size()` should have been used in order to allow the `ArrayList` to grow or shrink without causing any errors in the class. Some candidates did not understand that the *nextInt()* method in the `Random` class, when given a bound, will return a number from zero (inclusive) to the bound (exclusive). These candidates either put `answers.size()-1`, or the number 4, the size of the `ArrayList` minus one. Either answer would mean that the final entry in the `ArrayList` could never be returned. This is because if the bound given to *nextInt()* is *n*, then the number returned can be 0 to *n*–1 inclusive. Hence given the size of the `ArrayList` as the bound, the *nextInt()* method will return a number between 0 and the size –1, including 0 and size –1, and this corresponds to all the possible index values of items in the `ArrayList`.

Another way candidates showed their lack of understanding of what the *nextInt()* method returned, was by giving `answers.get(index-1);` for the second fragment. Since the *nextInt()* method can return 0, *index* could be zero, hence this statement has the potential to cause a run-time exception as follows: `Exception in thread "main" java.lang.IndexOutOfBoundsException: Index -1 out of bounds for length 5`

In part (iii) less than half of candidates attempting the question gave the correct answer to (A). In fact, the statement is *false* because an `ArrayList` cannot be parameterised to primitive variables. Perhaps the source of the confusion is that an `ArrayList` can contain primitive variables, if it is not parameterised, as the subject guide states in Volume 1, section 7.3:

> primitives must be 'wrapped' up as objects before they are inserted [into an `ArrayList`]. Wrapping and unwrapping are automatic in Java 5 and subsequent versions.

So, in a non-parameterised `ArrayList` any type of variable can be inserted, primitive or reference, and the `ArrayList` can contain different types of variables. For example in the *XXX* class the statement `a.add(5);` could be added without triggering any errors, and the `ArrayList` would then contain both a `String` and an `int`.

Most answers to (B) were correct, `ArrayLists` are about as fast as `Arrays` as section 7.3 in Volume 1 of the subject guide explains. All candidates answered (C) correctly.

**c. Answer**

```
/*1*/  this.input = new Scanner(input);
/*2*/  this.output = output;
/*3*/  this.wordsFilePath = wordsFilePath;
/*4*/  this.wordsFileCharset = wordsFileCharset;
/*5*/  random = new Random();
/*6*/  loadWordsOrGetDefaultWords();
```

**Comments**

Candidates were asked to write the body of the constructor in the *AbstractRandomWordGame* class, which should initialise all instance variables. Since there were six instance variables (and two *static* variables) the constructor should have had six statements, each initialising one of the instance variables.

In answering this question:

- The line numbered 1 in the above answer was problematic for candidates, with 60 per cent either giving `this.input = input;` as their answer, or not giving any statement to initialise the instance variable *input*.
- All candidates gave the lines numbered 2, 3 and 4 correctly.
- Line 5 was missing nearly half of the time, plus a few times `getRandomWord();` was included in the constructor instead of the correct line 5 statement to initialise the `Random` variable so that it could be used successfully to invoke the *Random.nextInt()* method in the *getRandomWord()* method.
- Line 6 was missing in about a third of answers, plus a small number of students wrote several statements into the constructor to try to initialise the *List* variable.

Line 1: The instance variable *input* is a `Scanner` variable, and it should be initialised by chaining it to a connection stream such as an `InputStream`. Therefore, the parameter *input*, which is of type `InputStream`, should have been used to initialise the `Scanner` object. Hence the answer `this.input=input;` gained no credit for that part of the question.

Lines 2–4 are more straightforward, since the parameter list contains parameters that are of the same type as the instance variables, so it is simply a matter of matching up the instance variable with the correct parameter.

If line 5 is missing or incorrect, when a subclass runs the *getRandomWord()* method a `NullPointerException` will be thrown.

Candidates were told in the question that the `List` variable *words* should:

> be initialised with words taken from a text file, or, if the file cannot be accessed for some reason, [should] be initialised to a default state.

Candidates were expected to read the class and understand that the *loadWordsOrGetDefaultWords()* method does these things, hence the constructor needs to invoke it.

## Question 3

This was not a popular question, attempted by about a quarter of candidates.

**a. Answer**

i. (A) When a class needs more than one `JButton`, in order to implement different actions when different `JButtons` are clicked, the accepted solution is to implement the Listener interface (e.g. `ActionListener`) once for each button using inner classes. **TRUE**

(B) Inner classes have access to all of their containing classes variables, except for private variables. **FALSE**

(C) Event sources (such as a `JButton`) can be registered with multiple event handlers. **TRUE**

(D) If a region is not specified then the *add()* method of `BorderLayout` will add the component to the CENTER region. **TRUE**

(E) The `BorderLayout` manager has 8 regions. **FALSE**

(F) `Swing` applications should not directly call the *paintComponent()* method **TRUE**

ii. (B) The top left corner.

**Comments**

Part (i) was answered correctly most of the time, with a few mistakes but no candidate giving less than five correct answers. The only common error seen was that a few candidates thought that (F) was *false*. Part (ii) was always answered correctly.

**b. Answer**

*Changes to the go() method:*

```
/*NEW*/ button.addActionListener(new ClickButtonListener());
/*3*/ - /*8*/   DELETED
```

*New inner class:*

```
class ClickButtonListener implements ActionListener{
   public void actionPerformed (ActionEvent e){
      button.setText("YEAH! I'VE BEEN CLICKED!");
      frame.repaint();
   }
}
```

**Comments**

This was mostly answered correctly, although a few candidates seemed to indicate in their answers that they thought the new inner class should be inside the *go()* method, which is not ideal.

**c. Answer**

```
class ShapesDrawPanel extends JPanel{
   public void paintComponent (Graphics g){

/*1*/  Graphics2D g2=(Graphics2D)g;
/*2*/  g2.setColor(Color.yellow);
/*3*/  g2.fillRect(0,0,this.getWidth(), this.getHeight());
/*4*/  g2.setColor(Color.blue);
/*5*/  g2.fillOval((this.getWidth()-(diameter+circleX))/2,
            (this.getHeight()-(diameter+circleX))/2,
              diameter+circleX, diameter+circleX);
/*6*/  g2.setColor(Color.red);
/*7*/  g2.fillRect(squareX,0,50,50);

         if ((circleX==0)|| (circleX==this.getHeight()-diameter)
            || (circleX==this.getWidth()-diameter))
              growCircle=!growCircle;
         if ((squareX==0)||squareX==this.getWidth()-50)
              moveSquare=!moveSquare;
         growCircle();
         moveSquare();
      }

      public void growCircle(){
            try{
              Thread.sleep(5);
            }
            catch (Exception ex){}
            if (growCircle)circleX++; else circleX--;
            repaint();
      }

      public void moveSquare(){
            try{
              Thread.sleep(5);
            }
            catch (Exception ex){}
            if (moveSquare)squareX++; else squareX--;
            repaint();
        }
} //text in bold given with the question
```

**Comments**

The model answer above is only one correct way of answering the question; there are others. All answers should deal with:

- detecting when the square has touched the right or left edge of the frame, and reversing its direction of movement once it has
- detecting when any part of the circle has touched the edge of the frame and causing it to start shrinking in size if it has

- detecting when the circle is back to its original size in order to start it growing again once it is.

The examiners were looking for answers with solutions to the above three problems that were logically correct in that they would detect a boundary condition and would then take correct action. An error often seen was that if the circle touched the edge of the frame it would start shrinking, and would continue shrinking until it disappeared as the candidate had no logic to detect when the circle was back to its original size and start it growing again.

Other errors were to do with taking appropriate action once a boundary was detected. One common logical error was detecting that the edge of the square had reached the right edge of the frame, and moving it back by one, but only by one. Hence the square would reach the edge and move back one, then move forward one, then move back one, move forward one, move back one, and this would continue indefinitely. Candidates making this mistake would write code that, in English, said "if the square has reached the left edge of the frame then deduct 1 from the variable giving its position on the frame, else add 1."

Despite some logical errors many correct answers were seen, written by students who were clearly very well prepared for the examination.

## Question 4

This question was attempted by 85 per cent of candidates.

**a. Answer**

i. (C)
```
10234
10234
```
ii. (C)
```
trainers
high jump
```
iii. Yes, mark the variable as *final*.

**Comments**

About 40 per cent of candidates gave the wrong answer to (i). (B) (`-19` and `10234`) was the most popular wrong answer, demonstrating that candidates did not understand that a *static* variable is the same for every object of the class. Hence the statement `count2.index = 10234;` changes the value of the *static index* variable to be 10234 for every object of the class.

More than half of candidates gave an incorrect answer to part (ii); incorrect answers varied indicating that candidates were most likely guessing. (C) is the correct answer because the *equipment()* method is a *static* method, and *static* methods can be invoked without an instance of the class, just as *equipment()* has been in the main method of the *Athletes* class. Almost all candidates answered (iii) correctly.

**b. Answer**

i.
```
15
20
greetings
to
all
```
ii. (B) `String s=String.format("%-20s %-20s", date, a);`

iii. `private StatsA(){}`

**Comments**

Part (i) was answered correctly by the majority, although some answers were seen that demonstrated that a few candidates had no idea how the `String.split()` method worked.

Nearly half of candidates gave incorrect answers to (ii). In answering this question candidates were expected to note that the output was a date, followed by the text assigned to `String` *a*. Hence statements (A) and (C) could not be correct since the output would be given with the variable *a* first. Hence the answer would have to be either (B) or (D). In choosing between (B) and (D) candidates were expected to note that (D) gave only 11 spaces to the *date* variable, meaning that the `String` variable's text would be printed without much space between it and the date. However (B) allows 20 spaces for the date, meaning that any spaces unused by printing the *date* variable would be added on to the end of the date, pushing the `String` variable's text further away and leaving a large gap between the date and the text "Today".

Most answers to part (iii) were correct, with the most common incorrect answer being public `StatsA(){}`. Note that classes with only static methods are often given a *private* constructor to prevent instantiation, as instantiation is not appropriate for classes with no instance variables and methods.

**c. Answer**

```
private static Person parsePerson(String line) {
   String[] details = line.split(", ");

   String name = details[0];
   String jobTitle = details[1];
   String teamName = details[2];
   int age = Integer.parseInt(details[3]);
   return new Person(name, jobTitle, teamName, age);
}


//alternative answer
private static Person parsePerson(String line) {
   String[] details = line.split(", ");
   return new Person(details[0].trim(),
      details[1].trim(), details[2].trim(),
         Integer.parseInt(details[3].trim()));
}
```

**Comments**

Many correct answers were given to part (c). A few mistakes were seen; some candidates assigned a `String` to the *age* variable without first parsing it to an `int`, which is quite a basic error. Another mistake was giving the method a `void` return type, with most candidates who made this error nevertheless returning a *Person*. This is a basic error; all candidates should understand that a method that returns a variable needs to be of the type that it is returning. Another error was made by candidates who thought that the *split()* method returned an `ArrayList`; these candidates wrote: `ArrayList<String> details = line.split(", ");`

The best answers seen used the `String.trim()` method on the `Array` items. This was very good, although not necessary for full credit.

## Question 5

About 60 per cent of candidates attempted this question.

**a. Answer**

i. (C) The path to a file.

ii. (A) Reads and writes bytes.

iii. (A) `BufferedReader` and `BufferedWriter` are connection streams. **FALSE**

(B) `BufferedReader` is efficient as it reads into a buffer and does not read from the file again until all the data in its buffer has been processed. **TRUE**

(C) `BufferedWriter` can be made to write to a file before its buffer is full by using its *flush()* method. **TRUE**

(D) `BufferedReader` can be chained to an `InputStreamReader` in order to read data from any source (e.g. the terminal, the internet, etc.). **TRUE**

**Comments**

Less than half of candidates gave the correct answer to part (i), with incorrect answers distributed fairly evenly over the other possible answers, suggesting that many candidates were guessing.

Only 25 per cent of candidates gave the correct answer to part (ii); most thought that the right answer was C: *Cannot be used to read or write binary data as it only reads or writes text.* In part (iii) most candidates thought that (A) was true; apart from this error, candidates answered the other statements correctly in most cases.

**b. Answer**

i. `u.openStream()`

ii. (A) It will print to standard output the IP address of the machine it is run on.

iii. (C) Serial version ID.

**Comments**

It was rare to see a correct answer to part (i). The most common answer given was `u,` with only 5 per cent of candidates giving the correct answer. The Java API for the `URL` class notes that the *openStream()* method:

> Opens a connection to this `URL` and returns an `InputStream` for reading from that connection. This method is a shorthand for:
>
> `openConnection().getInputStream()`

See: https://docs.oracle.com/javase/7/docs/api/java/net/URL.html#openStream()

Hence invoking the URL class's *openStream()* method is a sensible answer to this question as it will allow a connection to be made to the URL object, and the class to complete its task of reading from the web page given, and saving the text found there (i.e. the HTML code) to a new file.

Mostly answers to (ii) were correct, while answers to (iii) were fairly evenly divided between the correct answer and (B).

**c. Answer**

i. FileNotFoundException, IOException, ClassNotFoundException ii.

```
public static void serialize(ArrayList<String> results,
   File file) throws FileNotFoundException, IOException{
      FileOutputStream fos = new FileOutputStream(file);
      ObjectOutputStream oos=new ObjectOutputStream(fos);
      oos.writeObject(results);
      fos.close();
      oos.close();
} //bold text given with the question
```

**Comments**

In part (i) candidates were expected to understand that creating a `FileInputStream` could throw a `FileNotFoundException`, that creating an `ObjectInputStream` could cause an `IOException` and that `ObjectInputStream`'s *readObject()* method could throw both an `IOException`, *and* a `ClassNotFoundException`. In fact, most candidates gave at least one of `FileNotFoundException` and `IOException` in their answers, but less than 10 per cent of candidates gave the `ClassNotFoundException`. A few candidates gained some credit by giving answers such as *ClassException* or *ClassCastException* demonstrating that they understood that the deserializing process could throw an error if the JVM did not recognise the type of the deserialized class.

About a third of candidates either did not attempt an answer to part (ii), or gave answers that were very poor and showed no understanding of what was expected. Other candidates lost credit by giving answers that were copies of the *deserialize()* method, with every 'in' changed to 'out' (eg `ObjectInputStream` changed to `ObjectOutputStream`), and with *readObject()* changed to *writeObject()*.

Another, more minor error, was creating the `FileOutputStream` with a `String` file name, rather than with the `File` object given in the method's parameter list. This would work because the `FileOutputStream` class has a constructor that takes a `String` parameter; however candidates were expected to use, and not to ignore, the parameters of the method.

Overall, many attempts at Question 5 were quite poor.

## Question 6

About 55 per cent of candidates attempted this question.

**a. Answer**

i. (A) All exceptions can be caught with a single supertype catch. **TRUE**

(B) Java has checked and unchecked exceptions. Unchecked means that exceptions cannot be handled with try/catch. **FALSE**

(C) Checked exceptions are for run-time errors that are outside of the control of the developer. **TRUE**

(D) All unchecked exceptions subclass `java.lang.RuntimeException`. **TRUE**

(E) `ArrayIndexOutOfBoundsException` is an unchecked exception. **TRUE**

ii. (A) The compiler would report the following error:

```
error: exception UnsupportedEncodingException has
already been caught
```

**Comments**

In part (i) most candidates answered (A) and (E) correctly. About a quarter thought that (B) was true; to be clear all exceptions can be handled with try/catch, but the compiler *does not* enforce that unchecked exceptions are thrown or handled.

While most answers to (A), (B) and (E) were correct, more than 60 per cent of candidates thought that (C) was false. Checked exceptions are for situations that are outside of the developer's control. For example `IOException` is a checked exception, and can be thrown in many circumstances, including when a connection to a particular web page cannot be made using the `URL` class's *openStream()* method. In this case, it is reasonable to expect the developer to include try/catch blocks allowing recovery or a graceful exit.

Similarly, more than 60 per cent thought that (D) was false; these candidates are urged to visit https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html where they can read that:

> `RuntimeException` and its subclasses are *unchecked exceptions*. Unchecked exceptions do *not* need to be declared in a method or constructor's `throws` clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Readers might also want to consult the API for `Throwable`, the direct parent class of Exception, which states:

> The `Throwable` class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause. For the purposes of compile-time checking of exceptions, `Throwable` and any subclass of `Throwable` that is not also a subclass of either `RuntimeException` or `Error` are regarded as checked exceptions.
>
> See: https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

Finally, one candidate wrote a comment in answer to (D), stating that it was *false* as `Error` is unchecked and is a subclass of `Exception` (but not `RuntimeException`). However, this is only partly correct since `Error` is not an `Exception`, although it is treated as an unchecked exception by the compiler. `Error` is not an `Exception` because it is not a subclass of `Exception`. Like `Exception`, `Error` is a subclass of `Throwable`. The API for `Error` states that:

> An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. [...]
>
> A method is not required to declare in its `throws` clause any subclasses of `Error` that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur. That is, `Error` and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.
>
> See: https://docs.oracle.com/javase/7/docs/api/java/lang/Error.html

Part (ii) was answered correctly by less than half of candidates, with (B)*: The class would compile because the compiler does not check the order of catch blocks* the most popular wrong answer. Candidates were expected to understand that since catch blocks are visited by the JVM in the order that they are written, and since exceptions can be caught by catch blocks intended for one of their supertypes (superclasses), the compiler *does* expect them to be in a certain order, given by the inheritance hierarchy.

Since `UnsupportedEncodingException` is a child class of `IOException` (see [https://docs.oracle.com/javase/7/docs/api/java/io/UnsupportedEncodingException.html](https://docs.oracle.com/javase/7/docs/api/java/io/UnsupportedEncodingException.html)) an `IOException` catch block will handle an `UnsupportedEncodingException`. Hence if the `IOException` catch block is placed first, the compiler will flag this as an error.

Catch blocks for `Exception`, from which all exceptions inherit, should always be placed last. This is because it catches all of its subclasses, which means that it catches all exceptions. If an `Exception` catch block was placed first with any other catch block placed second, the compiler would flag that as an error.

**b. Answer**

i. Subclassing `Thread` (alternative answer: *using an anonymous class*).

ii. (D) All of the above

iii. *run()* and *call stack*, as follows: A `Runnable` object must have a run() method because this is defined by the `Runnable` interface. This method is placed on the bottom of the **call stack**.

**Comments**

Most answers given to (i) were wrong. A minority of candidates gave the answer *subclassing Thread*, no candidate gave the alternative answer above. Wrong answers that were given, such as `Thread t = new Thread();` did not explain a technique for making a `Thread`. The second volume of the subject guide, in Chapter 8, describes three techniques for making a `Thread`: implementing `Runnable`; subclassing `Thread` and overriding its *run()* method; and using an anonymous class.

In part (ii) very few candidates recognised that all the statements given were true. Candidates answered that either (B) or (C) were true, or that both (B) and (C) were true.

Few correct answers were given to (iii). Some answers were obvious guesses, with candidates offering Java key words such as *static* or *class*, together with `Thread`-related words such as *Runnable* or *Thread.* Many candidates understood that the missing text from the first blank space was *run(),* but guessed at the second, writing such things as *implementing class* or *Thread class* or *Runnable object.*

**c. Answer**

```
/*1*/ new Socket(host, port);
/*2*/ socket.getInputStream()
/*3*/ close();//flush() not accepted
/*4*/ close();
```

**Comments**

Very few (less than 20 per cent) completely correct answers were seen. Candidates often answered lines 1, 3 and 4 correctly, but found line 2 particularly problematic. There were no common errors for line 2; many different answers were seen. Most answers for line 2, such as *socket.accept()* and *socket.getHostName(),* were obvious guesses since they demonstrated no understanding. The majority of candidates answered correctly for line 3, although a few candidates answered `flush();` which is incorrect as `ObjectInputStream` does not have a *flush()* method, although `ObjectOutputStream` does. Line 4 was always correct.

Overall, this question was not attempted well by the majority of candidates.