Examiners' commentary 2018–2019

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

Comments on specific questions

Question 1

This question was attempted by 99 per cent of candidates.

a. Answer

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

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

Comments

Part (i) was answered correctly by most candidates, thus demonstrating their understanding that instance variables have default values until they are assigned. A few candidates lost some credit by writing only '0' and 'null', while others lost all credit by demonstrating their belief that there would be no output from the uninitialised instance variables, giving `Dog age is` as the first line and `Dog name is` as the second.

Part (ii) was almost always answered correctly, although a few candidates thought that the answer was (A), a few thought it was (B) while some thought the answer was both (B) and (C).

b. Answer

```
636 House 6
10 Woo Hoo!
10 House 6
8888 If not now then when?
```

Comments

This question tested candidates' understanding of:

- constructors
- inheritance
- the use of the keyword *super* to explicitly call a superclass constructor from a child class constructor
- that, failing an explicit call to a superclass constructor, the child class constructor makes an implicit one.

Some candidates answered this question very badly, an answer seen far too often was:

```
House 6 636
636
House 6
<blank line>
```

Candidates who gave this answer were copying from the paper the parameters given to the constructor of each object. Hence the first line comprises the parameters given to the constructor to make the *soh1* object, the second the single parameter for the *soh2* object, the third the single parameter for the *soh3* object, and the final line is blank because the *soh4* object did not have any parameters. Candidates giving this answer were making no effort at all to trace the construction of each object through the constructors in the *SonOfHoo* and *Hoo* classes. For example, the *soh2* object is created with an `int` parameter, and this will invoke the second constructor in the *SonOfHoo* class. This parameter calls the superclass parameter with a `String` parameter with the statement: `super("Woo Hoo!");`. The statement will invoke the second constructor in the *Hoo* class, since it takes a single `String` parameter. This constructor will assign the `String` parameter "Woo Hoo!" to the `String` instance variable *s* and 10 to the `int` instance variable *i*. The original parameter given to the *SonOfHoo* constructor, 636, is not used.

Note that each constructor in *SonOfHoo* includes an explicit call to one of the superclass constructors, except for the no-argument constructor. In this case the call to the superclass constructor is implicit, hence the compiler would insert `super();` invoking the superclass' no-argument constructor.

Most candidates thought that there would be no output from the final `System.out.println()` statement. Better answers were that the output would be `0` and `null`, but only about one third of candidates knew that the implicit call to the superclass constructor would result in the fields of the *soh4* object being assigned the default values given by *Hoo*'s no-argument constructor.

Many candidates were able to give the correct output for the first three lines, but often with the fields in the wrong order (e.g. `Woo Hoo! 10` instead of `10 Woo Hoo!`), or perhaps with each field printed on a new line. These candidates were ignoring the *toString()* method in the *Hoo* class, which the JVM would use to print the *SonOfHoo* objects. The *toString()* method prints the fields on one line, in the order `int, String`.

c. Answer

```
public class Robot extends Machine{

   public Robot (String name, int pin){
      super(name, pin);
   }

   public void broadcast(String s){
      System.out.println(s);
   }

(i)   public void speak(){
         System.out.println("I am your robot overlord");
      }
(ii)  public void broadcast (){
         System.out.println("the robot uprising has
         begun");
      }
}//bold text given with the question
```

Comments

In part (i) most candidates gave the correct answer, with the remainder giving *speak()* a `String` parameter, which was an error since 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.

Part (ii) was usually answered correctly, but with two errors seen. Some candidates wrote the method with a `String` parameter, and were given no credit since overloaded methods must have different parameter lists. Others lost some credit by giving their method parameters, often a `String` and an `int`. While this is overloading, the parameters were pointless as they were not needed or used in the method, and would make invoking the method more difficult since users would have to think up arbitrary values for them.

Question 2

This question was attempted by about 66 per cent of candidates.

a. Answer

i. (A) An abstract class cannot be instantiated **TRUE**

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

(C) An abstract class can have concrete (i.e. non-abstract) methods. **TRUE**

(D) The abstract key word is used in an abstract class's declaration for convenience, it is not necessary as the compiler can detect that the class is abstract. **FALSE**

ii.

```
public class IntStuff extends Ex{
   public boolean lessThanTen(int x, int y){
      int i = x+y;
      return(i<10);
   }
}//other correct answers are possible
```

Comments

Overall, part (i) was answered correctly by most candidates, except that candidates varied in their answer to (B). The expected answer to (B) was *true*, because a class can only extend one class. However, the examiners thought that some candidates were answering *false* because the wording of the question was unclear, and implied that a class could always extend one abstract class, irrespective of whether or not the class extended any concrete classes. The examiners decided to accept *true* or *false* as the correct answer to (B).

Part (ii) was usually answered correctly, although a minority of candidates left the *abstract* keyword in the heading of their otherwise correct methods, and a few candidates wrote a `void` method that nevertheless returned a `boolean` value, a basic error. All candidates should know that a method can only return a variable or value of its type. A void method cannot return anything.

b. Answer

i. `/*3*/ String s = a.get(0);`

ii.

```
public static String magic8Ball() {
      Random r = new Random();
      int index = r.nextInt(answers.size());
      return answers.get(index);
}
```

iii. (A) `Arraylists` can be parameterised to object variables, but not to primitive variables. **TRUE**

(B) `ArrayLists` use random access and are about as fast as `Arrays`. **TRUE**

(C) The enhanced for loop can be used for iterating through what Java calls collections. It cannot be used with an `ArrayList` because Java does not consider the class to be a part of the collections framework. **FALSE**

Comments

Most candidates answered part (i) correctly, but almost half thought that line 1 was the problem, or that lines 1, 2 and 3 would all give compilation errors. These candidates thought that an `ArrayList` could not be declared without a parameter, as happened in line 1: `ArrayList a = new ArrayList();`. With non-parameterised `ArrayLists` all entries are objects. Primitives can be added to a non-parameterised `ArrayList`, but they are first wrapped into their reference type, for example `int` to `Integer`, by the JVM. All entries have to be cast to their type when being extracted from the `ArrayList`. Hence line 3 is the single line that would cause a compilation error, and could be corrected by casting the retrieved value as follows: `String s = (String)a.get(0);`

One wrong answer seen more than once was that line 4 would give a compilation error because there was no spacing around '='. This was a basic error that could only have been made by candidates who had practised very little or no programming. There are some tokens that the compiler would need spaces between in order to understand them as separate entities, for example `String s` would be understood as a `String` variable called `s`, while `Strings` would give a compilation error. But the compiler does not need spaces around the equals sign, since it knows that the equals sign is an assignment operator in Java, and it will always treat it as such.

In part (ii) all candidates gained at least some credit for this question, since mostly correct answers were seen. However, many candidates lost some credit by putting '5' for the first missing fragment, when `answers.size()` should have been used in order to allow the `ArrayList` to grow or shrink in later versions of the *HB* class, without causing any errors. Some candidates put the number '4', just as some put `answers.size()-1`. These 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). Hence given 4 (or equivalently `answers.size()-1`) the *nextInt()* method could return 0, 1, 2, or 3, meaning that the final value in the `ArrayList`, indexed by 4, could never be returned.

Part (iii) was answered correctly by most candidates, with some confusion about (A). One candidate wrote that the answer was *false*, as autoboxing could be used. This candidate was perhaps confused, as were others, between adding primitive variables to an unparameterised `ArrayList` (possible) and using a primitive type as a parameter (not possible). For example `ArrayList<int> test=new ArrayList<int>();` would be flagged as an error by the compiler with: `error: unexpected type`.

When primitive values are added to an unparameterised `ArrayList`, they are inserted as objects, hence they are automatically converted to objects, for example a `boolean` would be wrapped to a `Boolean`. This wrapping is done automatically since Java 5, and is known as autoboxing.

c. Answer

```
public AbstractRandomWordGame(InputStream input,
   PrintStream output, Path wordsFilePath, Charset
   wordsFileCharset) {
      this.input = new Scanner(input);
      this.output = output;
      this.wordsFilePath = wordsFilePath;
      this.wordsFileCharset = wordsFileCharset;
      random = new Random();
      loadWordsOrGetDefaultWords();
}//text in bold given with the question
```

Comments

A minority of candidates answered this question correctly. A typical answer to this question would be:

```
this.input = input;
this.output = output;
this.wordsFilePath = wordsFilePath;
this.wordsFileCharset = wordsFileCharset;
```

Hence the most common errors were:

- `this.input = input;`. Since *input* the instance variable is a `Scanner` object, and *input* the parameter is an `InputStream` object, this will be flagged by the compiler with: `error: incompatible types: InputStream cannot be converted to Scanner.`
- The `Random` instance variable was not initialised, hence when a subclass runs the *getRandomWord()* method a `NullPointerException` will be thrown.
- The `List` instance variable was not initialised. Candidates were told in the question that the `List` variable *words* should be initialised by reading from a text file, but if the file was not available for some reason, then it should be initialised to a default state. Candidates were expected to read the *AbstractRandomWordGame* class and understand that the *loadWordsOrGetDefaultWords()* method does these things, hence the constructor should invoke it.

Question 3

This question was attempted by around one third 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 have the Listener call back method (e.g. *actionPerformed()*) query the event source. **FALSE**

(B) Inner classes have access to all of their containing classes' variables, including private variables. **TRUE**

(C) Event sources (such as a `JButton`) can only be registered with one event handler. **FALSE**

(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 3 regions. **FALSE**

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

ii. (A) The top left corner.

Comments

Part (i) was answered mostly correctly, with the only common error being that most candidates thought that (A) was *true*. Part (ii) was always answered correctly.

b. Answer

Changes to the go() method:

```
/*NEW*/  button.addActionListener(new ButtonListener());
/*3*/ - /*8*/  Lines 3 – 8 deleted.
```

New inner class:

```
class ButtonListener implements ActionListener{
   public void actionPerformed (ActionEvent e){
      button.setText("I said here not there!");
      frame.repaint();
   }
}
```

Comments

This was mostly answered correctly, although a few candidates missed `implements ActionListener` from their inner class declaration, and a minority of candidates seemed to think that the inner class should be inside the *go()* method, which it should not be.

c. Answer

```
class ShapesDrawPanel extends JPanel{
   public void paintComponent (Graphics g){
/*1*/ - /*7*/  LINES 1 – 7 INCLUDED

   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 was given with the question

Comments

The examiners were looking for correct logic in answer to this question, and ignored minor syntax errors. All answers should detect when the shapes were approaching a boundary condition, which for the square would mean touching the left or right edge of the frame, and for the circle would mean touching the frame or reaching its starting size while shrinking. Clearly the logic to detect and take action when a boundary condition is reached could be implemented in different ways.

Common logical errors seen were: taking action when one boundary condition was reached, but not the other; using the wrong variable to control the animation of one or other of the shapes (for example changing the *diameter* of the circle rather than *circleX*); or implementing the change of direction of the animation incorrectly.

By far the most common logical error was in implementing the change of direction. Some candidates attempted to change direction depending on the value of a `boolean` variable that was never updated appropriately when a boundary was reached. Some candidates wrote `while` loops for the animation, with a `boolean` guard that was not updated, hence the loop would be infinite. In addition, writing loops inside *paintComponent()* is problematic, and unnecessary. The most common error seen was that some candidates attempted to change direction by writing code that, in English, said "if the shape has reached a boundary then deduct 1 from the variable giving its position (or size) on the frame, else add 1." This would mean that the shape would reach the boundary, shrink or move back by one, then move back again to the boundary, move back by one, move forward by one, and that this would continue indefinitely.

Some candidates added new variables for the animation, failing to understand that it was the *circleX* variable that should be used to grow and shrink the circle, and the *squareX* variable should be used to move the square. These candidates often struggled to write logically correct animation methods and statements. One mistake often seen was incrementing the new variable used for animation, until the boundary was reached, then multiplying the variable by –1, presumably intended to change the direction of the animation, but actually giving the shape a negative position or size parameter. For example, the circle might be given its size by the new variable, and this variable might reach 500 say, and then be changed to –500.

Question 4

This question was attempted by 99 per cent of candidates.

a. Answer

i. (B)
```
pole
pole
```

ii. (A)
```
Hello and welcome
5
```

iii. Yes, mark the method as *final*.

Comments

Most candidates gave the wrong answer to (i). (A) (`trainers` and `pole`) 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 `athlete2.equipment = pole;` changes the value of the *static equipment* variable to be *pole* for every object of the class.

Most candidates gave the correct answer to part (ii) demonstrating that they understood that a static method could be invoked without an object of the class, but (iii) was more problematic, with many wrong answers seen, most of them different (e.g. make abstract; make static; make the method private; method hiding), indicating a lot of guessing.

b. Answer

i.
```
16
32 //end of the output of the first for loop
hello
and
welcome //end of the output of the second loop
```

ii. (D)

iii.
```
private StatsB(){}
```

Comments

Parts (i) and (ii) were answered correctly by most candidates, while part (iii) was answered correctly by only a minority. Many different answers were seen, such as:

- `static StatsB(){}`
- `public abstract StatsB(){}`
- `static abstract StatsB(){}`
- `final StatsB(){}`
- `final public StatsB(){}`
- `public StatsB(){}`

Also, quite a number of candidates made no attempt at this question. 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].trim();
   String jobTitle = details[1].trim();
   String teamName = details[2].trim();
   int age = Integer.parseInt(details[3].trim());
   return new Person(name, jobTitle, teamName, age);
}
            String s = String.format("%-11s %-7s",date,a);

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

Comments

A large minority wrote methods that returned a *Person*, but gave their methods a different type: `void`, `String` and `ArrayList<Person>` were all seen by the examiners, as were methods with no type at all. All candidates should understand that a method that returns a variable needs to be of the type that it is returning. Another common error was made by candidates who thought that the *split()* method returned an `ArrayList`; these candidates wrote: `ArrayList<String> details = line.split(", ");`. Some candidates correctly invoked `String.split()` but lost marks by using `ArrayList` syntax, such as round brackets around the index numbers or using the `ArrayList.get()` method, for example `String name = details.get(0);`.

Other very basic errors were assigning `details[3]` to the *age* variable without first parsing it to an `int`, and copying the `String` method parameter, in the above the *line* variable, to a local parameter. The latter is not a syntax or logic error, merely unnecessary.

A very small number of candidates made their *Person* object by invoking a no-argument constructor, and directly assigning values to the fields of the object; for example `Person p = new Person(); p.name = details[0];` even though (1) the *Person* class does not have a no-argument constructor and (2) the *Person* class's instance variables are private, and so cannot be accessed outside the class.

One candidate commented that the *readPeople()* method should not be *private*, which is correct, it should be *public* to allow other classes to use the method. 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

This question was attempted by 55 per cent of candidates.

a. Answer

i. (A) The path to a file.

ii. (C) Cannot directly read input or write output but must be connected to another stream that can.

iii. (A) `BufferedReader` and `BufferedWriter` are chain streams. **TRUE**

(B) `BufferedWriter` is efficient as it fills its buffer before writing to the file, reducing the number of times the file is written to. **TRUE**

(C) The only way to make `BufferedWriter` write to a file before its buffer is full, is to close the stream. **FALSE**

(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

Most candidates gave the answer to part (i) as (B) or (C) while a few answered (D). Less than half of candidates gave the correct answer.

In part (ii) a large minority thought that the answer was (A) or (B). In part (iii) statement (B) was the only one that was almost always answered correctly. A large minority gave an incorrect answer to at least one of the other statements.

b. Answer

i. `u.openStream()`

ii. (B) None of the above

iii. (A) Serial version ID.

Comments

The most common answer given for part (i) was `u,` with about 20 per cent of candidates giving the correct answer. Many obvious guesses were seen, for example, `InetAddress.getByName(u)` and `getHostAddress(u),` most likely copied from the *HI* class in part (ii). Other answers, such as `u.getInputStream()` at least showed understanding that a connection should be made to the `URL` object, `u`.

Although (A) and (D) were popular (wrong) answers, most answers to (ii) were correct, while answers to (iii) were usually correct, with (B) the most likely incorrect answer.

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);
   os.writeObject(results);
   fos.close();
   oos.close();
}//bold text given with the question
```

Comments

In part (i) the majority of candidates gave only `FileInputStream` (could be thrown by creating a `FileInputStream` object) and `IOException` as their answer. Hence, most candidates did not know that `ObjectInputStream`'s *readObject()* method could throw both an `IOException`, *and* a `ClassNotFoundException`. An `IOException` could also be thrown when creating an `ObjectInputStream`.

Part (ii) was usually answered correctly with a few mistakes seen. A small minority tried to write to the file in a loop, with one entry from the `ArrayList` serialized with each iteration. Candidates were expected to read the *SerializationUtil* class and understand that since the *deserialize()* method was deserializing an `ArrayList<String>` the *serialize()* method should therefore serialize an `ArrayList<String>`.

A few candidates lost credit by copying the statement to deserialize from the *deserialize()* method, as in the following in which *in* has been changed to *out* and *read* to *write*:

```
results = (ArrayList<String>) out.writeObject();
```

A small minority copied the entire *deserialize()* method, changing *in* to *out* throughout, and *readObject* to *writeObject*. These candidates received no credit.

Question 6

This question was attempted by 45 per cent of candidates.

a. Answer

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

(B) Unchecked means that the compiler does not check that the exceptions have been handled in the code. **TRUE**

(C) Unchecked exceptions can generally be avoided by good programming logic. **TRUE**

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

(E) `FileNotFoundException` is an unchecked exception. **FALSE**

ii. (B) The compiler would report the following error:
```
error: exception FileNotFoundException has already been caught
```

Comments

In part (i) most candidates answered correctly, with the only common error being that (D) was often thought to be false. The API for `Throwable`, the direct parent class of Exception, states:

> 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

Hence unchecked exceptions subclass `RuntimeException`, and `Error` (a subclass of `Throwable` as is `Exception`) is treated like an unchecked exception by the compiler, but is not an exception.

A minority answered part (ii) correctly. The most popular wrong answers were (A): *The class would compile because the compiler does not check the order of catch blocks* and (C): *The class would compile but when run would stop immediately with a run-time error*. Candidates who answered (C) at least appreciated that there was an issue with the order of catch blocks. 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.

Catch blocks for `Exception` 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.

Since `UnsupportedEncodingException` is a child class of `IOException` (see 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.

b. Answer

i. Implementing the `Runnable` interface (alternative answer: *using an anonymous class*).

ii. (D) All of the above.

iii. *start()* and *run()* as follows: When your code calls the start() method on the `Thread` object, the `Thread` object invokes run() on its `Runnable` instance variable.

Comments

Most answers given to (i) were wrong, and many different answers were seen such as *use synchronization* or *superclassing* that demonstrated no understanding or knowledge. A minority of candidates gave the answer *Implementing the Runnable interface*; no candidate gave the alternative answer above. Candidates who wrote that an alternative way to make a `Thread` would be to implement an interface or similar were given some credit for knowing that the answer involved an interface. 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) most candidates recognised that all the statements given were true, and so answered (D).

About 30 per cent of candidates answered (iii) correctly. Some candidates understood that the second missing text was *run(),* but did not know what the first was, and gave answers that were clearly guesses, such as *new, blocked* or *go()*.

c. Answer

```
/*1*/ new Socket(host, port);
/*2*/ socket.getInputStream()
/*3*/ close();
/*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, with the most common wrong answer being *socket*. 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.