

1 (8 marks)

- (a) A *static* method is invoked on a(n)

Answer: class, or class name

Marking scheme: 4 marks for a correct answer, 0 marks otherwise.

- (b) A *non-static* method is invoked on a(n)

Answer: object, or instance, or reference

Marking scheme: 4 marks for a correct answer, 0 marks otherwise.

2 (20 marks)

- (a) What is the *state* of an object?

Answer: The values of all of its attributes.

Marking scheme: 4 marks for a correct answer, 0 marks otherwise.

- (b) What is the *identity* of an object?

Answer: Its address, or some other unique identifier.

Marking scheme: 4 marks for a correct answer, 0 marks otherwise.

- (c) What does the binary operator `==` check?

Answer: Equality of identity, or equality of value.

Marking scheme: 4 marks for a correct answer, 0 marks otherwise.

- (d) What does the method `equals` check?

Answer: Equality of state.

Marking scheme: 4 marks for a correct answer, 0 marks otherwise.

- (e) Consider the following fragment of Java code:

```
1 Fraction f1 = new Fraction(1, 2);
2 Fraction f2 = new Fraction(3, 4);
3 Fraction f3 = f1;
```

Circle the expressions below that evaluate to `true`:

`f1 == f1`

`f1 == f2`

`f2 == f3`

`f1 == f3`

`f1.equals(f1)`

`f1.equals(f2)`

`f2.equals(f3)`

`f1.equals(f3)`

Answers: Underlined above.

Marking scheme: 1 mark for each correct answer, -1 for each incorrect answer. Lowest possible total mark is 0.

3 (8 marks)

What are the *two* differences between a **Set** and a **List**?

Answer: No duplicate elements allowed (**Set**) versus duplicate elements allowed (**List**). Unordered (**Set**) versus sequential order (**List**); or only iterator-based access (**Set**) versus indexed-based access (**List**).

Marking scheme: 4 marks for each of the two differences.

4 (28 marks)

Consider the following fragment of (legal) Java code:

```
1  Number num = new BigInteger("123456789123456789");
2  double val = num.doubleValue();
3  num = ((BigInteger) num).subtract(new BigInteger("1"));
```

BigInteger extends **Number**.

The method **doubleValue()** is polymorphic.

The cast on line 3 is necessary for the code to compile.

- (a) What class or classes are searched during early binding for the method **doubleValue()**? **Justify your answer.**

Answer: The declared type of **num** is **Number**; thus, early binding searches **Number**.

Marking scheme:

- 2 marks for mentioning the declared type
- 2 marks for the correct class (**Number**).

- (b) What class or classes are searched during early binding for the method **subtract(BigInteger)**? **Justify your answer.**

Answer: The declared type of **num** is **Number**, which is cast to **BigInteger** for the invocation; thus, early binding searches **BigInteger**.

Marking scheme:

- 2 marks for mentioning the declared type
- 4 marks for mentioning the cast
- 2 marks for the correct class (**BigInteger**).

- (c) What class or classes are searched during late binding for the method **doubleValue()**? **Justify your answer.**

Answer: The actual (or runtime) type of **num** is **BigInteger**; thus, late binding searches **BigInteger**.

Marking scheme:

- 6 marks for mentioning “actual type”
- 6 marks for identifying the actual type as `BigInteger`
- 4 marks for the correct class (`BigInteger`).

5 (4 marks)

Explain how composition differs from aggregation.

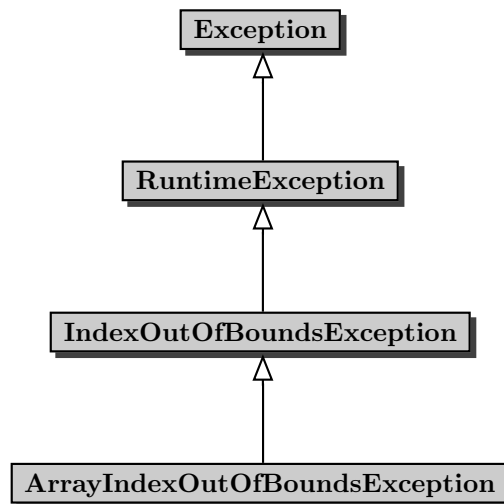
Answer: Composition is a stronger form of aggregation where the lifetime of the object and its aggregated objects are the same.

Alternative answer: In a composition, the whole cannot exist without its parts. In an aggregation, the whole and the parts can exist without each other.

Marking scheme: 4 marks for a correct answer.

6 (28 marks)

Consider the following UML diagram.



(a) Consider the following main method.

```

1 public static void main(String[] args)
2 {
3     PrintStream output = System.out;
4
5     try
6     {
7         output.println(args[-1]);

```

```

8      ...
9      }
10     catch (ArrayIndexOutOfBoundsException e)
11     {
12         output.println("Invalid array index is used");
13     }
14     catch (Exception e)
15     {
16         output.println("Something went wrong!");
17     }
18 }

```

The above main method produces the following output.

`Invalid array index is used`

Explain what happens when the code is run. In your explanation, use the following phrases if they are relevant (not all may be relevant):

- throw(s) an exception
- terminate(s) immediately
- determine(s) the appropriate
- substitutable/substitutability

Answer: On line 7, `args[-1]` throws an exception of type `ArrayIndexOutOfBoundsException`. Execution of the `try` block terminates immediately. The virtual machine determines that the appropriate exception handler is `catch (ArrayIndexOutOfBoundsException e)` on line 10. Program flow goes to the handler which produces the given output.

Marking scheme:

- 4 marks for correctly using “throws an exception” (or its equivalent)
- 4 marks for correctly using “terminates immediately” (or its equivalent)
- 4 marks for correctly using “determines the appropriate” (or its equivalent)

(b) Consider the following main method.

```

1  public static void main(String[] args)
2  {
3      PrintStream output = System.out;
4
5      try
6      {
7          output.println(args[-1]);

```

```

8      ...
9    }
10   catch (IndexOutOfBoundsException e)
11   {
12       output.println("Invalid index is used");
13   }
14   catch (Exception e)
15   {
16       output.println("Something went wrong!");
17   }
18 }

```

The above main method produces the following output.

Invalid index is used

Explain what happens when the code is run. In your explanation, use the following phrases if they are relevant (not all may be relevant):

- throw(s) an exception
- terminate(s) immediately
- determine(s) the appropriate
- substitutable/substitutability

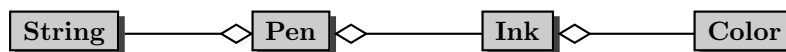
Answer: On line 7, `args[-1]` throws an exception of type `ArrayIndexOutOfBoundsException`. Execution of the `try` block terminates immediately. The virtual machine determines that the appropriate exception handler is `catch (IndexOutOfBoundsException e)` on line 10 because `ArrayIndexOutOfBoundsException` is substitutable for `IndexOutOfBoundsException`. Program flow goes to the handler which produces the given output.

Marking scheme:

- 4 marks for correctly using “throws an exception” (or its equivalent)
- 4 marks for correctly using “terminates immediately” (or its equivalent)
- 4 marks for correctly using “determines the appropriate” (or its equivalent)
- 4 marks for correctly using “substitutable” (or its equivalent)

7 (14 marks)

Consider the following UML diagram.



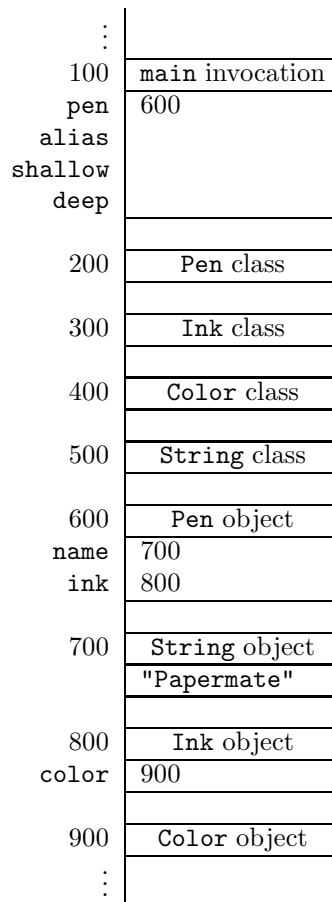
Consider the following code snippet (which appears in the body of a main method).

```

1 Pen pen = new Pen("Papermate", new Ink(Color.BLUE));
2 Pen alias = ...;
3 Pen shallow = ...;
4 Pen deep = ...;

```

Assume that memory can be depicted by the diagram below when the execution reaches the end of line 1. Recall that the **String** class is immutable.



- (a) Consider that an *alias* of **pen** is assigned to **alias** in line 2. Draw the diagram depicting memory when the execution reaches the end of line 2. Draw only those blocks which have changed or are new.

Answer:

⋮	
100	main invocation
pen	600
alias	600
shallow	
deep	
⋮	

Marking scheme: 2 marks if `alias` has the address 600 as its value, 0 marks otherwise.

- (b) Consider that a *shallow* copy of `pen` is assigned to `shallow` in line 3. Draw the diagram depicting memory when the execution reaches the end of line 3. *Draw only those blocks which have changed or are new.*

Answer:

⋮	
100	main invocation
pen	600
alias	600
shallow	1000
deep	
1000	Pen object
name	700
ink	800
⋮	

Marking scheme:

- 2 marks if `shallow` has a new (one that was not used before) address as its value, 0 marks otherwise.
 - 2 marks if there a new `Pen` object at that new address, 0 marks otherwise.
 - 1 mark if the value of `name` of the new `Pen` object is 700, 0 marks otherwise.
 - 1 mark if the value of `ink` of the new `Pen` object is 800, 0 marks otherwise.
- (c) Consider that a *deep* copy of `pen` is assigned to `deep` in line 4. Draw the diagram depicting memory when the execution reaches the end of line 4. *Draw only those blocks which have changed or are new.*

Answer:

⋮	
100	main invocation
pen	600
alias	600
shallow	1000
deep	1100
1100	Pen object
name	700
ink	1200
1200	Ink object
color	1300
1300	Color object
⋮	

Marking scheme:

- 2 marks if **deep** has a new (one that was not used before) address as its value, 0 marks otherwise.
- 2 marks if there a new **Pen** object at that new address, 0 marks otherwise.
- 1 mark if the value of **name** of the new **Pen** object is 700, 0 marks otherwise. (Here you use the fact that the class **String** is immutable and, hence, there is no point to make a copy.)
- 1 mark if the value of **ink** of the new **Pen** object is the address of a new **Ink** object, 0 marks otherwise.

8 (10 marks)

(a) What is a loop invariant?

Answer: something that holds at the end of every iteration (page 193 of the textbook).

Marking scheme:

- Something like “a boolean expression (or something) that holds at the beginning (or end) of every iteration” deserves 2 marks.
- An answer that contains “boolean expression” but does that mention “every iteration” deserves 1 mark.

- (b) Searching a collection for a particular element is performed using a loop. Suppose that we want to know whether or not a list contains at least one string that starts with **z**. Consider the following attempt:

```
1 // t is a List<String>
2 boolean found = false;
3 for (int i = 0; i < t.size(); i++)
4 {
5     found = t.get(i).startsWith("z");
6 }
```

What is a useful loop invariant in the code fragment shown above?

Answer: `found == (i == 0)  $\vee$  t.get(i - 1).startsWith("z")`

Alternative answer: the value of `found` matches the value of “the last element visited thus far starts with a z.”

Marking scheme:

- A correct and useful loop invariant deserves 2 marks.
- If the loop invariant is incorrect but mentions the last element, then 1 mark.

- (c) The loop invariant in Part (b) does not correctly solve the search problem. What is a useful loop invariant that correctly solves the search problem?

Answer: `found ==  $\exists 0 \leq j < i : t.get(j).startsWith("z")$`

Alternative answer: the value of `found` matches the value of “one of the elements visited thus far starts with a z.”

Marking scheme:

- A correct and useful loop invariant deserves 2 marks.
- If the loop invariant is incorrect but mentions that some element starts with z, then 1 mark.

- (d) Using big-O notation, state the complexity of the loop shown in Part (b) where N is the number of elements in the list.

Answer: $O(N)$.

Marking scheme: 2 marks if $O(N)$, 0 marks otherwise.

- (e) Suppose we want to solve a different search problem: Does a set contain all of the elements of another set? Consider the following attempt:

```

1  // set1 is a Set<String>
2  // set2 is a Set<String>
3  boolean containsAll = true;
4  for (String s1 : set1)
5  {
6      boolean found = false;
7      for (String s2 : set2)
8      {
9          found = found || s1.equals(s2);
10     }
11     containsAll = containsAll && found;
12 }

```

Using big-O notation, state the complexity of the code fragment shown above. Explain your answer (but do not formally prove your answer). You can assume that both sets always have N elements.

Answer: $O(N^2)$.

Marking scheme: 2 marks if $O(N^2)$, 0 marks otherwise.