BIRKBECK

(University of London)

MSc EXAMINATION FOR INTERNAL STUDENTS

MSc Computer Science
MSc Learning Technologies
MSc Computing for the Financial Services

Department of Computer Science and Information Systems

Programming in Java

BUCI033S7

DATE OF EXAMINATION: Wednesday, 25th May 2016
TIME OF EXAMINATION: 14:30–17:30
DURATION OF PAPER: Three hours

WITH OUTLINE SOLUTIONS

RUBRIC:

1. Candidates should attempt ALL 10 questions on this paper.
2. You are advised to look through the entire examination paper before getting started, in order to plan your strategy.
3. Simplicity and clarity of expression in your answers is important.
4. All programming questions should be answered using the JAVA programming language unless stated to the contrary.
5. Electronic calculators **ARE** allowed.
6. START EACH QUESTION ON A NEW PAGE.
7. Extracts from the following Java APIs are included as Appendix A:
    - `java.util.Comparator<T>`
    - `java.util.Collections`
    - `java.util.List<E>`

Question:	1	2	3	4	5	6	7	8	9	10	Total
Marks:	4	15	8	7	10	13	8	11	9	15	100

Question 1 .... Total: 4 marks

Suppose you have a class `Person` with `private` instance variables `firstName` and `lastName`, both of type `String`. Provide the following methods for the class `Person` which override the default methods from `Object`:

(a) `toString` returns a persons full name.

(b) `equals` returns true if another `Person` object has the same first and last names as the given `Person`.

Solution:

```java
public class Person {
    private String firstName;
    private String lastName;

    @Override
    public String toString() {
        // The answer can equally well just use string concatenation
        StringBuilder s = new StringBuilder();
        s.append("First:␣").append(firstName)
                .append("Surname:␣").append(lastName);
        return s.toString();
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Person)) return false;
        if (o == this) return true;
        Person p = (Person) o;
        return (firstName.equals(p.firstName)
                && lastName.equals(p.lastName));
    }
}
```

1 mark for `toString` and 3 marks for `equals`.

Question 2 .... Total: 15 marks

For each of the following Java language features, give an example of a place in a large Java library where you might expect it to be used. Explain exactly what the feature achieves, why that is a benefit in the context you describe, and what risk or inconvenience might arise if the feature were not deployed.

(a) Methods that have been declared as `protected`. 3 marks

(b) Classes that are labelled as `final`. 3 marks

(c) Generic methods – that is, ones where the types of their arguments and results involve other types enclosed in angle brackets, as in `ClassName<AnotherClassName>`. 3 marks

(d) Fields within a class that are marked as `private`. 3 marks

(e) Parts of the library defined as an `interface` rather than as a `class`. 3 marks

Solution: Most of this question is bookwork…

(a) Protected access for methods would enable the subclasses to utilise the methods without exposing them to clients. This might be for the purpose of reusing functionality. Otherwise one might have to duplicate code.

(b) Final for a class would prohibit sub-typing. This may be a problem if one has not thought through the use cases.

(c) Generic methods places constraints on the method usage for the types that it may be used with but again, the constraints may be inflexible or too general, if the use cases are not clearly thought through.

(d) Enables encapsulation but prohibits access to subclasses or package which may be undesirable in certain use cases.

(e) The interface approach enables efficient decoupling of specification from implementation; the coupling can then be provided at a later stage (perhaps utilising dependency injection or a factory). A downside might be if you wish to use the interface as a *mixin* (for code reuse) but this is less of a problem now with the advent of Java 8.

Question 3 .... Total: 8 marks

Consider the following method:

```java
import java.util.LinkedList;
import java.util.List;

public class Mystery {
    public List<Integer> mystery(final int[][] data) {
        final List<Integer> result = new LinkedList<>();
        for (int i = 0; i < data.length; i++) {
            int sum = 0;
            for (int j = 0; j < data[i].length; j++) {
                sum = sum + j * data[i][j];
            }
            result.add(sum);
        }
        return result;
    }
}
```

What function does the code compute? Explain your answer by tracing the following inputs to the function and stating the output:

(a) `[[3, 4], [1, 2, 3], [], [5, 6]]`

(b) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

Solution: Sums the sublists returning them as a list.

With the following outputs:

(a) `[4, 8, 0, 6]`

(b) `[8, 17, 26]`

Question 4 .... Total: 7 marks

(a) Briefly describe four distinct reasons as to why you should not commit compiled code (such as `.class` files) to a version control repository. 4 marks

Solution: Any four from the following although other answers may be equally acceptable:

1. Merge conflicts cannot be resolved. Another way of saying the same thing is that binary files are not diff-able (by the standard text-based diff algorithms).

2. Repetition of information in source and binary forms violates the DRY (dont repeat yourself) principle.

3. Binary files such as `.class` files are architecture-dependent and may not be useful to others.

4. Binary files may contain information such as timestamps that is guaranteed to create a conflict even if generated from the same source code by others.

5. Bloat in the VCS because differences are huge.

6. Timestamps might not be preserved.

7. If there is a check-in without compiling, then they can be inconsistent with the source code.

(b) "It is cheaper and faster to fix known bugs before you write new code." 3 marks

Do you agree? Briefly provide three reasons to support your statement. Provide reasons that are as different from one another as possible.

Solution:

- You are familiar with the code now. A related reason is that the bug will be harder to find and fix later.

- Later code may depend on this code. A related reason is that a bug may reveal a fundamental problem.

- Leaving all bugs to the end will make it harder to understand and keep to the schedule, because its hard to predict how long bug fixing will take.

- An overfull bug database is demoralising and is likely to be ignored.

- You will be able to add tests for the bug once its been fixed to avoid future issues.

- Avoid feature creep.

- ...

Question 5 .... Total: 10 marks

Using the fragments below:

```
<Item>         <MyGenerics>   <Integer>      <>
Item           MyGenerics     Integer        Float
compare        sort           p              c
one.size       two.size       x.size         p.size
```

complete the following code so that it compiles and prints

```
4 3 2 1
```

Note, you may use a fragment any number of times, including zero.

```
import java.util.*;

public class MyGenerics {
  List ① p = new ArrayList ② ();

  class Comp implements Comparator<Item> {
     public int ③(Item one, Item two){
       return ④ − ⑤;
     }
  }

  class Item {
    int size;
    Item(int s){
      size = s;
    }
  }

  public static void main(String[] args) {
    new MyGenerics().go();
  }

  void go(){
    p.add(new Item(4));
    p.add(new Item(1));
    p.add(new Item(3));
    p.add(new Item(2));

    Comparator<Item> c = new Comp();
    Collections.sort( ⑥ , ⑦ );
    for( ⑧ x : p)
      System.out.print( ⑨ + "␣");
  }
}
```

You do not need to copy the program in your answer; just indicate which fragment should be substituted for each of ① through ⑨.

See Appendix A for the abbreviated Javadoc for `Comparator`, `Collections`, and `List`.

Solution:

① `<Item>`

② `<>`

③ `compare`

④ `two.size`

⑤ `one.size`

⑥ `p`

⑦ `c`

⑧ `Item`

⑨ `x.size`

1 mark for each slot + 0.5 mark each for getting p/c and two/one.size in the correct order.

Question 6 . . . . Total: 13 marks

Consider the following Java class definition :

```
public class OutOfRangeException extends Exception {
    private static final long serialVersionUID = 1L;
    private final String reason;

    public OutOfRangeException(final String reason) {
        this.reason = reason;
    }

    public String getReason() {
        return reason;
    }
}
```

(a) Provide the definition of a Java class `Interval` which contains a constructor taking two integer parameters `lower` and `upper`; if `lower` exceeds `upper` then it should throw an `OutOfRangeException` exception, otherwise it should save these values in two member variables; 4 marks

(b) a method `in`, which takes an integer parameter `num` and throws an exception of type `OutOfRangeException` if `num` is not between the `lower` and `upper` bounds; 3 marks

(c) a static method `testInterval` which takes three integer parameters `low`, `high`, and `val`, and constructs the interval `a` by calling 6 marks

```
new Interval(low,high)
```

It then checks whether `val` is within the interval `a` by calling `a.in(val)`. The method should handle any exceptions that are thrown by printing the reason.

Solution:

```
public class Interval {
    private int lower;
    private int upper;

    // part (i)
    public Interval(final int lower, final int upper)
            throws OutOfRangeException {
        if (lower > upper) {
            throw new OutOfRangeException("lower exceeds upper");
        }
        this.lower = lower;
        this.upper = upper;
    }

    // part (iii)
    public static void testInterval(int low, int high, int val) {
        Interval a;
        try {
            a = new Interval(low, high);
            a.in(val);
        } catch (OutOfRangeException ex) {
            System.out.println(ex.getReason());
        }
    }

    // part (ii)
    public void in(final int num) throws OutOfRangeException {
        if (lower > num || num > upper) {
            throw new OutOfRangeException(”parameter␣out␣of␣range”);
        }
    }
}
```

Question 7 . . . . Total: 8 marks

(a) What is meant by the term *generic* in the context of Java? Explain the main purpose of generics and the most important syntax associated with them. 5 marks

Solution: Bookwork but should provide appropriate examples to illustrate their answer.

(b) One of your fellow students puts forward the proposition “`String` is a sub-class of `Object`, therefore `ArrayList<Object>` is a sub-class of `ArrayList<String>`”. Discuss this proposition. 3 marks

Solution: Should indicate that while `String` and `Object` are covariant on type this does not apply to the container type `ArrayList<T>`. They do not have to use the term *covariant* but should clearly explain what the problem is.

Question 8 . . . . Total: 11 marks

Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.

For example:

```
permute("TEAM")
```

outputs the following sequence:

```
TEAM TEMA TAEM TAME TMEA TMAE ETAM ETMA EATM EAMT EMTA EMAT
ATEM ATME AETM AEMT AMTE AMET MTEA MTAE META MEAT MATE MAET
```

Solution: Although a recursive solution is provided here an iterative one is equally acceptable.

```
public class Permute {
    private Permute() {
    }

    public static void main(final String[] args) {
        permute(”TEAM”);
    }

    // Outputs all permutations of the given string.
    public static void permute(final String s) {
        permute(s, "");
    }

    private static void permute(String s, String chosen) {
        if (s.length() == 0) {
            // base case: no choices left to be made
            System.out.println(chosen);
        } else {
            // recursive case: choose each possible next letter
            for (int i = 0; i < s.length(); i++) {
                final char c = s.charAt(i);                 // choose
                s = s.substring(0, i) + s.substring(i + 1);
                chosen += c;
                permute(s, chosen); // explore
                // deselect
                s = s.substring(0, i) + c + s.substring(i);
                chosen = chosen.substring(0, chosen.length() - 1);
            }
        }
    }
}
```

Question 9 . . . . Total: 9 marks

Given the following class:

```
public class Strange {
    private final static int TEN = 10;
    private final static int HUNDRED = TEN * TEN;

    public static void strange(final int n) {
        if (n < 0) {
            System.out.print(”−”);
            strange(−n);
        } else if (n < TEN) {
            System.out.println(n);
        } else {
            final int two = n % HUNDRED;
            System.out.print(two / TEN);
            System.out.print(two % TEN);
            strange(n / HUNDRED);
        }
    }
}
```

For each of the following calls to the method `strange`, state what value is returned and why:

(a) `strange(7)`,

(b) `strange(825)`,

(c) `strange(38947)`.

Solution:

(a) 7

(b) 258

(c) 47893

+ appropriate discussion of how they came about the answers showing that they understand how the recursive routine works.

Question 10 . . . . Total: 15 marks

(a) Streams in Java 8 defer execution of most operations until the results are actually required. There are three types of methods that operate with these *lazy* streams. State each of the types with an example of each. 5 marks

Solution:

Intermediate methods — These are methods that produce other Streams. These methods dont get processed until there is some terminal method called. (1 mark)

Terminal methods — After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it. These methods can result in a side-effect (`forEach`) or produce a value (`findFirst`). (2 marks)

Short-circuit methods — These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated. Short-circuit methods can be intermediate (`limit`, `substream`) or terminal (`findFirst`, `allMatch`). (2 marks)

including appropriate examples

(b) What is the role of the `Optional` class in Java? You should provide an appropriate example to illustrate your answer. 3 marks

Solution: An `Optional<T>` either stores a `T` or stores nothing. This is useful for methods that may or may not find a value and where one wants to avoid returning `null` so that computations can behave correctly without throwing exceptions or explicitly checking for `null` values.
+ any appropriate example

(c) Create a `static` method called `transformedList` of the class `StringUtils` which has the following formal parameters: 7 marks

- a `List` of `String`s, and
- a `Function<String,String>`

and returns a new `List` that contains the results of applying the function to each element of the original list. E.g.:

- `List<String> excitingWords =`
  `StringUtils.transformedList(words, s -> s + "!");`
- `List<String> eyeWords =`
  `StringUtils.transformedList(words, s -> s.replace("i", "eye"));`
- `List<String> upperCaseWords =`
  `StringUtils.transformedList(words, String::toUpperCase);`

You should not use any existing *higher-order-functions* available from the standard Java libraries.

Solution:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

public class StringUtils {
    public static List<String> transformedList(List<String> originals,
                                               Function<String, String> tra
        List<String> results = new ArrayList<>();
        for (String original : originals) {
            results.add(transformer.apply(original));
        }
        return results;
    }

    public static void main(String[] args) {
        List<String> words = Arrays.asList(”a”, ”b”, ”i”, ”c”);
        System.out.println(transformedList(words, s -> s + ”!”));
        System.out.println(transformedList(words, s -> s.replace(”i”, ”eye”
        System.out.println(transformedList(words, String::toUpperCase));
    }
}
```

Appendix A: Extracts from various APIs

`java.util.Comparator<T>`

`abstract int`	`compare(T lhs, T rhs)` Compares the two specified objects to determine their relative ordering. Returns an integer < 0 if `lhs` is less than `rhs`, 0 if they are equal, and > 0 if `lhs` is greater than `rhs`.
`abstract boolean`	`equals(Object object)` Compares this `Comparator` with the specified `Object` and indicates whether they are equal. Returns boolean `true` if specified `Object` is the same as this `Object`, and `false` otherwise.

`java.util.Collections`

`static List`	`EMPTY_LIST` The empty list (immutable).
`static Map`	`EMPTY_MAP` The empty map (immutable).
`static Set`	`EMPTY_SET` The empty set (immutable).
`static <T> boolean`	`addAll(Collection<? super T> c, T... elements)` Adds all of the specified elements to the specified collection.
`static <T> void`	`copy(List<? super T> dest, List<? extends T> src)` Copies all of the elements from one list into another.
`static <T> List<T>`	`emptyList()` Returns an empty list (immutable).
`static <K,V> Map<K,V>`	`emptyMap()` Returns an empty map (immutable).
`static <T> Set<T>`	`emptySet()` Returns an empty set (immutable).
`static <T> boolean`	`replaceAll(List<T> list, T oldVal, T newVal)` Replaces all occurrences of one specified value in a list with another.
`static <T> void`	`sort(List<T> list, Comparator<? super T> c)` Sorts the specified list according to the order induced by the specified comparator.

java.util.List<E>

boolean	add(E e) Appends the specified element to the end of this list (optional operation).
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
void	clear() Removes all of the elements from this list (optional operation).
boolean	contains(Object o) Returns true if this list contains the specified element.
boolean	equals(Object o) Compares the specified object with this list for equality.
E	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or −1 if this list does not contain the element.
boolean	isEmpty() Returns true if this list contains no elements.
E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
default void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.