BIRKBECK

(University of London)

MSc EXAMINATION FOR INTERNAL STUDENTS

MSc Computer Science

Department of Computer Science and Information Systems

Programming in Java

BUCI033S7

DATE OF EXAMINATION: Friday, 9 June 2017
TIME OF EXAMINATION: 10am
DURATION OF PAPER: Three hours

WITH OUTLINE SOLUTIONS

RUBRIC:

1. Candidates should attempt ALL 11 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	11	Total
Marks:	9	6	6	9	5	16	8	8	9	9	15	100

Question 1 ..... Total: 9 marks

For each of the following statements indicate whether it is *true* or *false*.
There is one mark for each correct answer.
Please note: Incorrect answers will be penalised.

(a) **F** A public top-level Java class may be defined in a source file with any base name as long as the file extension is `.java`.

(b) **F** Java identifiers can contain letters, digits, and the underscore symbol and may start with a digit.

(c) **F** The statement `int x = 3f/4f;` will compile, but the result will be truncated so that `x` gets the value `0`.

(d) **F** In a for loop header, `for (`*initialiser*`;` *condition*`;` *update*`)`, the Java compiler requires *initialiser* to initialise a loop variable and *update* to update it.

(e) **T** The declarations `double[] scores` and `double... scores` are equivalent.

(f) **F** Java arrays have a variable number of elements of mixed types.

(g) **F** Given an array named `scores`, the statement `scores[scores.length + 1]` will not compile.

(h) **F** Instance methods of a class can be called without first instantiating an object of the class.

(i) **F** Every Java class has a default no-arg constructor in addition to the constructors you write yourself.

Solution: ½ mark deducted for an incorrect answer with a minimum of zero overall.

Question 2 ..... Total: 6 marks

Briefly explain the following terms:

(a) abstract class, 3 marks

(b) interface. 3 marks

Use appropriate examples to illustrate your answer. Include in your answers the use of the keywords `extends` and `implements`, and indicate where they are (and are not) applicable.

Solution: An abstract class is one that is not used to construct objects, but only as a basis for making subclasses.

An interface consists of a set of instance method interfaces, without any associated implementations. (Can also contain constants.)

Should use `extends` for extending an interface or concrete class; `implements` for instantiation of an interface as a (possibly abstract) class.

Question 3 ..... Total: 6 marks

How does method overloading differ from method overriding? Provide appropriate examples to illustrate your answer.

Solution: Roughly three marks each including appropriate examples.

method overloading same method name with a different signature.

method overriding method with same signature replaced in subclass.

Question 4 ..... Total: 9 marks

This question concerns Unit testing and TDD.

(a) What is Unit testing? Provide a brief example to illustrate your answer. [3 marks]

Solution: A unit test is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.

(b) When should Unit testing be carried out? [1 mark]

Solution: At each "save iteration" and certainly before Integration testing.

(c) Who should carry out Unit testing? [1 mark]

Solution: Unit testing should be done by the developers.

(d) State four advantages of carrying out Unit testing. [4 marks]

Solution: Something like the following but other answers maybe equally acceptable.

1. Issues are found at early stage. Since unit testing are carried out by developers where they test their individual code before the integration. Hence the issues can be found very early and can be resolved then and there without impacting the other piece of codes.
2. Unit testing helps in maintaining and changing the code. This is possible by making the codes less interdependent so that unit testing can be executed. Hence chances of impact of changes to any other code gets reduced.
3. Since the bugs are found early in unit testing hence it also helps in reducing the cost of bug fixes. Just imagine the cost of bug found during the later stages of development like during system testing or during acceptance testing.
4. Unit testing helps in simplifying the debugging process. If suppose a test fails then only latest changes made in code needs to be debugged.

Question 5 ..... Total: 5 marks

State five reasons as to why one should prefer composition over inheritance. For each reason you should include an example to support your answer.

Solution: One mark for each reason.

1. One reason of favouring *Composition* over *Inheritance* in Java is that Java doesn't support multiple inheritance of implementation. Since you can only extend one class in Java, but if you need multiple functionality, e.g., reading and writing character data into a file, you need `Reader` and `Writer` functionality and having them as private members makes things easier; which is composition. If you are following best practice and providing an interface rather than an implementation, and using the type of the base class as a member variable, you can use different `Reader` and `Writer` implementations for different situations. You don't get this flexibility by using inheritance; in the case of extending a class, you only get the facilities which are available at compile time.

2. Composition offers better test-ability of a class as compared to inheritance. If one class is composed of another class, you can easily create a Mock Object representing the composed class for testing. Inheritance doesn't provide this feature. To test the derived class, you will need its super class. Since unit testing is important during software development, composition wins over inheritance.

3. Many object oriented design patterns mentioned by the *Gang of Four* favour *Composition over Inheritance*. An example of this is the *Strategy* design pattern, where composition and delegation is used to change behaviour, without touching the target object's code. As we are using composition to hold the strategy, instead of obtaining it via inheritance, its easy to provide a new strategy implementation at run-time.

4. Though both Composition and Inheritance allows you to reuse code, one of the disadvantages of inheritance is that it breaks encapsulation. If sub-class depends on a super-class behaviour for its operation, it suddenly becomes fragile. When the behaviour of the super-class changes, functionality in the sub-class(es) may get broken, without any change on its part.

5. If you use composition you are flexible enough to replace the implementation of a composed class with a better and improved version. One example is using the `Comparator` class which provides `compare` functionality. If your object contains a `Comparator` instead of extending a particular `Comparator` for comparing , its easier to change the way comparison is performed by setting a different type of `Comparator` in the composed instance, while in case of inheritance you can only have one comparison behaviour, and you cannot change it at runtime.

Question 6 ..... Total: 16 marks

Consider the following class definition:

```
public abstract class Date {
    private int day;          // from 1 to 31
    private int month;        // from 1 to 12
    private int year;         // from 2000 upwards

    public void advance() {
    }  // "advance" to next day
}
```

(a) Implement a constructor that initialises new objects of the `Date` class to be set to the 1st of January 2000. 3 marks

(b) Implement getters and setters for each of the private fields. 2 marks

(c) Provide appropriate `hashCode`, `toString`, and `equals` methods for the class. 6 marks

(d) Provide an implementation for the `advance` method which advances the date to the next day. You should ensure that all the private fields are updated appropriately using their respective getters and setters. (You may assume that all months have the same number of days.) 5 marks

Solution:

```
package datesol;

public abstract class Date {
    private int day;          // from 1 to 31
    private int month;        // from 1 to 12
    private int year;         // from 2000 upwards

    public Date() {
        this(1, 1, 2000);
    }

    public Date(int day, int month, int year) {
        setDay(day);
        setMonth(month);
        setYear(year);
    }

    // "advance" to next day
    public void advance() {
        if (day < 31) {
            setDay(day + 1);
        } else {
            if (month < 12) {
                setDay(1);
                setMonth(month + 1);
            } else {
                setYear(year + 1);
```



```
                setMonth(1);
                setDay(1);
            }
        }
    }

    public int getDay() {
        return day;
    }

    public void setDay(int day) {
        this.day = day;
    }

    public int getMonth() {
        return month;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    @Override
    public String toString() {
        return "Day " + day + " Month " + month + " Year " + year;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (this == obj) return true;
        if (!(obj instanceof Date)) return false;
        Date d = (Date) obj;
        return this.getDay() == d.getDay()
                && this.getMonth() == d.getMonth()
                && this.getYear() == d.getYear();
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
```

```
        result = prime * result + getDay();
        result = prime * result + getMonth();
        result = prime * result + getYear();
        return result;
    }
}
```

Question 7 ..... Total: 8 marks

Provide two example implementations of a stack, one using inheritance the other using composition. You should use a `java.util.List` in your answer.

Solution: Extending `ArrayList` weakens the encapsulation of the class. Methods to get and insert elements at specific positions are available to clients of the stack. We move from trying to create a well behaved LIFO stack to creating a (possibly) leaking structure: an `EvilStack`.

```
package stack;

import java.util.ArrayList;

public class Stack<T> {
    private ArrayList<T> items = new ArrayList<>();

    public T pop() {
        return items.remove(items.size() - 1);
    }

    public void push(T item) {
        items.add(item);
    }
}
```

```
package stack;

import java.util.ArrayList;

public class EvilStack<T> extends ArrayList<T> {
    public T pop() {
        return remove(size() - 1);
    }

    public void push(T item) {
        add(item);
    }
}
```

Question 8 ..... Total: 8 marks

Write three versions of a method that calculates the factorial of a number. The three versions should be:

(a) An iterative version. 2 marks

(b) A recursive version. 3 marks

(c) A version which utilises the functional features of Java 8 (i.e., *streams*). 3 marks

Solution:

```
package factorial;

import java.util.stream.LongStream;

public class Factorial {

    public static void main(String... args) {
        System.out.println("The factorial of 10 is (iterative) "
                + iterative(10));
        System.out.println("The factorial of 10 is (recursive) "
                + recursive(10));
        System.out.println("The factorial of 10 is (streams) "
                + streams(10));
    }

    private static long iterative(long number) {
        long fact = 1;

        for (long i = 1; i <= number; i++) {
            fact = fact * i;
        }
        return fact;
    }

    private static long recursive(long number) {
        if (number == 0)
            return 1;
        else
            return (number * recursive(number - 1));
    }

    private static long streams(long number) {
        return LongStream.rangeClosed(2, number).reduce(1, (a, b) -> a * b);
    }
}
```

Question 9 ..... Total: 9 marks

Using the fragments below:

abstract	abstract	class	interface
compare	while	for	instanceof
Object	int	double	boolean
array	Integer	Double	Boolean
throw	throws	new	return
public	protected	final	static

complete the following code so that it compiles and prints the contents of the various arrays.

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

```
1  ① ② AbstractSort {
2
3    /* Extract an object from a given array */
4    ④ probe(③[] array, ⑤ index) {
5      m.probeCount++;
6      return ⑥[index];
7    }
8
9    /* Compare two objects */
10   int compare(Object a, Object b) {
11     if ((a ⑦ Integer) && (b ⑧ ⑨)) {
12       return compareInteger(a,b);
13     } else {
14       ⑩
15       new UnsupportedOperationException(
16         "Comparing between this type is not supported");
17     }
18   }
19
20   /* Compare two integers using Integer.compareTo */
21   int compareInteger(Object a, ❶ b) {
22     m.compareCount++;
23     ❷ ((Integer)a).compareTo((Integer)b);
24   }
25
26   /* Move an object inside an array using an offset */
27   void move(❸[] ❹, int index, int offset) {
28     m.moveCount++;
29     array[index + offset] = array[index];
30   }
31
32   /* Set an object into an array at a given index */
33   void set(Object[] array, ❺ index, ❻ value) {
34     m.setCount++;
35     ❼[index] = value;
36   }
37
38   AbstractSort(Metrics m) {
39     //this.m = m;
40   }
```

```
41
42    ❽ void report() {
43        System.out.println("Total probe count: " + m.probeCount);
44        System.out.println("Total compare count: " + m.compareCount);
45        System.out.println("Total move count: " + m.moveCount);
46        System.out.println("Total set count: " + m.setCount);
47    }
48
49    private final Metrics m = new Metrics();
50    private Object[] array;
51 }
```

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

Solution: 0.5 marks for each one.

① — `abstract`

② — `class`

③ — `Object`

④ — `Object`

⑤ — `int`

⑥ — `array`

⑦ — `instanceof`

⑧ — `instanceof`

⑨ — `Integer`

⑩ — `throw`

❶ — `Object`

❷ — `return`

❸ — `Object`

❹ — `array`

❺ — `int`

❻ — `Object`

❼ — `array`

❽ — `public`

Question 10 . . . . . Total: 9 marks

What output is produced when the following Java program is executed?
You should show your workings via a trace of the program.

```
1  public class Trace {
2      public static void main(String[] args) {
3          A a = new A();
4          B b = new B();
5          A ba = (A) b;
6
7          a.f(a);
8          a.f(b);
9          b.f(a);
10         b.f(b);
11         a.f(ba);
12         b.f(ba);
13         ba.f(a);
14         ba.f(b);
15         ba.f(ba);
16     }
17 }
18
19 class A {
20     public void f(A a) {
21         System.out.println("fa(A)");
22     }
23
24     public void f(B b) {
25         System.out.println("fa(B)");
26     }
27 }
28
29 class B extends A {
30     public void f(A a) {
31         System.out.println("fb(A)");
32     }
33
34     public void f(B b) {
35         System.out.println("fb(B)");
36     }
37 }
```

Solution:

```
fa(A)
fa(B)
fb(A)
fb(B)
fa(A)
fb(A)
```

```
fb(A)
fb(B)
fb(A)

```

+ trace.

Question 11 ..... Total: 15 marks

(a) Two threads each hold a resource that the other is requesting. This is an example of: 2 marks

A. Deadlock
B. Resource contention
C. Livelock
D. Race condition
E. Resource lock

(b) When a program's result relies upon the execution order of a program's threads, it is said to contain a: 2 marks

A. Deadlock
B. Timing bug
C. Livelock
D. Race condition

(c) Java contains built-in support for writing threaded programs. Which of the following are examples of this support? 2 marks

A. The `synchronized` keyword.
B. The `for` loop.
C. The `Thread` class.
D. The `private` keyword.
E. The `super` keyword.

(d) Answer the following questions with either true or false. No explanation necessary. There is one mark for each correct answer. 5 marks
Please note: Incorrect answers will be penalised.

i. **F** Threads cannot access objects that were created by a different thread.
ii. **F** When two threads simultaneously call the same method, one thread may overwrite the values of the local variables of that method from the other thread.
iii. **F** A Java program ends when the thread that executed `main()` terminates.
iv. **F** If you run a Java program on a computer with eight cores, you can create at most eight threads.
v. **F** One starts a Java thread by calling the `run()` method.

Solution: ½ mark deducted for an incorrect answer with a minimum of zero overall.

(e) The following code may or may not be correct. It was written by someone who is looking to make a counter class which is shareable between many threads. If it is correct, state why. If it is incorrect, fix it. 4 marks

```
public class ShareableCounter {
    private int i;

    public ShareableCounter() {
        i = 0;
    }

    public int inc() {
```

```
        i = i + 1;
        return i;
    }
}
```

Solution:

```
package thread;

public class ShareableCounter {
    private int i;

    public ShareableCounter() {
        i = 0;
    }

    public synchronized int inc() {
        i = i + 1;
        return i;
    }

// Or put a synchronized block for this around the i=i+1.

}
```

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 obj)` Compares this `Comparator` with the specified `Object` and indicates whether they are equal. Returns boolean `true` if the 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.