BIRKBECK

(University of London)

BSc EXAMINATION FOR INTERNAL STUDENTS

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

Software and Programming III

BUCI056H6

CREDIT VALUE: 15 credits

DATE OF EXAMINATION: Tuesday, 6 June 2017
TIME OF EXAMINATION: 10am
DURATION OF PAPER: Three hours

RUBRIC

1. **Candidates should attempt ALL questions in the paper.**
2. **The number of marks varies from question to question.**
3. **You are advised to look through the entire examination paper before getting started, in order to plan your strategy.**
4. **Simplicity and clarity of expression in your answers is important.**
5. **Electronic calculators are NOT permitted.**
6. **You may answer questions using only the Scala programming language unless specified otherwise.**
7. **You must not use mutable state (`vars`) or mutable collections in your solutions unless specified otherwise.**
8. **The last page of this exam contains an appendix which is useful for formulating your solutions. You can detach this page and keep it aside.**
9. **Start each question on a new page.**

Question:	1	2	3	4	5	6	7	8	Total
Marks:	11	8	10	19	16	14	16	6	100

Question 1 . . . . . Total: *11 marks*

Briefly explain the following terms:

(a) Inheritance — 4 marks

(b) Encapsulation — 3 marks

(c) Referential Transparency — 4 marks

with respect to object-oriented programming. You should use appropriate examples to fully illustrate your answers.

Question 2 . . . . . Total: *8 marks*

What do you understand by the following functional programming terms:

(a) Currying — 4 marks

(b) Closure — 4 marks

You should provide appropriate examples to illustrate your answers.

Question 3 . . . . . Total: *10 marks*

This question concerns the *SOLID* design principles.

(a) The *Interface Segregation* and *Dependency Inversion* principles are both about defining interfaces. They address different problems, however. Explain that difference in one or two sentences. — 2 marks

(b) A system to print-staple-fold-fax etc. has one main `Job` class that is used by most other tasks. During the development of the system, making a modification to the `Job` class affected many other classes and extended the compilation time. — 8 marks

A refinement to the system design was made to take the big `Job` class, which had many methods specific to a variety of different clients, and introduce multiple smaller interfaces for each client. A client would then only operate on a specific, small interface to the `Job` class. Modifications to the `Job` class would then affect a smaller number of interfaces and therefore less classes had to be recompiled.

Several SOLID principles may have been violated by the original design and by the refinement. Comment on which principles have been broken, why, and how you would redesign the system to adhere to SOLID principles?

Question 4 . . . . . Total: *19 marks*

(a) The Strategy and Command patterns both suggest using objects in place of methods. By the use of appropriate examples explain what the difference is between these two patterns? — 5 marks

(b) The Composite pattern and the Decorator pattern are similar. Describe the differences between the two and give examples of when you would use each. — 5 marks

(c) If you need to add behaviour to an object without changing its class which design patterns might you use? Provide an example illustrating the use of the design pattern. — 4 marks

(d) You are required to connect two parallel class hierarchies by letting subclasses in one hierarchy determine which class to instantiate in the second hierarchy. Which software design pattern should you use and why? — 5 marks

Question 5 . . . . . Total: *16 marks*

Suppose we have an application where we consume an `EmailService` to send emails. We could implement this service as:

```
case class EmailService() {
  def sendEmail(message: String, receiver: String) =
    println("Email sent to " + receiver + " with Message=" + message)
}
```

The `EmailService` class holds the logic to send an email message to the recipient email address. Our application code might be:

```
case class MyApplication() {
  private val email = EmailService()

  def processMessages(msg: String, rec: String) = {
    //do some msg validation, manipulation logic, etc.
    email.sendEmail(msg, rec)
  }
}
```

Our client code that will use `MyApp` to send email messages might be:

```
object MyDIClient extends App {
  val app = MyApplication()
  app.processMessages("Hi Fred", "fred@dcs.bbk.ac.uk")
}
```

(a) At first glance there appears to be nothing wrong with above implementation (it does compile and run!) but above code has certain limitations. Briefly describe each of these limitations. **6 marks**

(b) Now apply the dependency injection pattern to solve the problems with above implementation. Your code will need to adhere to the following requirements: **10 marks**

- Service components should be designed with a base class or an interface.
- Consumers should be written in terms of the service interface.
- Provide an injector class that will initialise the services and then the consumers.

Question 6 . . . . . Total: *14 marks*

(a) The reflection API allows an executing SCALA program to examine or "introspect" upon itself. Write a method which uses reflection to find out (and output) which methods are defined within a class. You should include the formal parameter and return types in your output together with any checked exceptions the methods may throw. **10 marks**

(b) Given the following class, what would the output be when your code is run on the class? **4 marks**

```
class Sample {
  @throws[NullPointerException]
  private def method1(p: Any, x: Int) = {
    if (p == null) throw new NullPointerException
    x
  }
}
```


Question 7 . . . . . Total: *16 marks*

(a) Provide an implementation of the following `differences` function, which takes a list of integers, and returns a list of pairwise differences of the elements of this list: **6 marks**

```
def differences(ls: List[Int]): List[Int] = ???
```

Specifically, calling `differences(xs)` must return a list `ys` such that:

- `ys.size == ls.size`
- If `ls.nonEmpty` then `ys.head == ls.head`
- For all `0 < i < ys.size`, `ys(i) == xs(i) - xs(i - 1)`

Some examples of usage are:

```
scala> differences(Nil)
res0: List[Int] = Nil

scala> differences(List(1))
res1: List[Int] = List(1)

scala> differences(List(1, -2, 3, -4, 5, -6))
res2: List[Int] = List(1, -3, 5, -7, 9, -11)
```

(b) Now implement the `rebuildList` function, which takes a list corresponding to the list of differences, and rebuilds the original list from it. For all `xs: List[Int]`, `rebuildList(differences(xs)) == xs`: **10 marks**

```
def rebuildList(ls: List[Int]): List[Int] = ???
```

Some examples of usage are:

```
scala> rebuildList(differences(Nil))
res3: List[Int] = Nil

scala> rebuildList(differences(List(1)))
res4: List[Int] = List(1)

scala> rebuildList(differences(List(1, -2, 3, -4, 5, -6)))
res5: List[Int] = List(1, -2, 3, -4, 5, -6)
```

You may make use of the API shown in Appendix A.

Question 8 . . . . . Total: *6 marks*

Consider the following Scala code:

```
1  package traits
2
3  class A {
4    def bar() = ""
5  }
6
7  trait B extends A {
8    override def bar() = super.bar() + "B"
9  }
10
11 trait C extends B {
12   override def bar() = super.bar() + "C"
13 }
14
15 trait D extends B {
16   override def bar() = super.bar() + "D"
17 }
18
19 object Main extends App {
20   foo(new A with D with C with B())
21
22   def foo(x: A with D) {
23     println(x.bar())
24   }
25 }
```

When the program is executed what would the output be? You should clearly explain your answer via a trace utilising the line numbers on the code listing.

Appendix A Scala Standard Library Methods

Here are some methods from the Scala standard library that you may find useful, on `List[A]`:

- `xs ++ (ys: List[A]): List[A]` — appends the list `ys` to the right of `xs`, returning a `List[A]`.
- `xs.apply(n: Int): A, or xs(n: Int)` — returns the `n`-th element of `xs`. Throws an exception if there is no element at that index.
- `xs.drop(n: Int): List[A]` — returns a `List[A]` that contains all elements of `xs` except the first `n` ones. If there are less than `n` elements in `xs`, returns the empty list.
- `xs.filter(p: A => Boolean)` — List[A]: returns all elements from xs that satisfy the predicate p as a List[A].
- `xs.flatMap[B](f: A => List[B]): List[B]` — applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
- `xs.foldLeft[B](z: B)(op: (B, A) => B): B` — applies the binary operator op to a start value and all elements of the list, going left to right.
- `xs.foldRight[B](z: B)(op: (A, B) => B): B` — applies the binary operator op to a start value and all elements of the list, going right to left.
- `xs.map[B](f: A => B): List[B]` — applies f to every element of the list xs and returns a new list of type List[B].
- `xs.nonEmpty: Boolean` — returns true if the list has at least one element, false otherwise.
- `xs.reverse: List[A]` — reverses the elements of the list xs.
- `xs.scan[B >: A](z: B)(op: (B, B) => B): List[B]` — produces a List[B] containing cumulative results of applying the operator op going left to right, with the start value z. The returning list contains one more element than the input list, the head being z itself.
- `xs.take(n: Int): List[A]` — returns a List[A] containing the first n elements of xs. If there are less than n elements in xs, returns these elements.
- `xs.zip(ys: List[B]): List[(A, B)]` — zips elements of xs and ys in a pairwise fashion. If one list is longer than the other one, remaining elements are discarded. Returns a List[(A, B)].

You can use the same API for `Stream`, replacing `List` by `Stream`. Stream (containing elements of type A):

- `xs #:: (ys: => Stream[A]): Stream[A]` — Builds a new stream starting with the element `xs`, and whose future elements will be those of `ys`.

