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: Friday, 8th June 2018
TIME OF EXAMINATION: 10:00am
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. You may answer questions using only the Scala or Kotlin programming languages unless specified otherwise.
6. You should avoid the use of mutable state (`vars`) or mutable collections in your solutions unless specified otherwise.
7. The last few pages of this exam contain appendices which may prove useful for formulating your solutions.
8. Start each question on a new page.

Question:	1	2	3	4	5	6	7	8	Total
Marks:	13	13	7	15	8	15	14	15	100

Question 1 . . . . Total: *13 marks*

(a) With reference to the object-oriented model of programming, what is reflection? Provide an example to illustrate your answer. **[2 marks]**

(b) Provide three examples where reflection is used. **[3 marks]**

(c) Is a program's execution speed slower if it utilises reflection? Justify your answer. **[2 marks]**

(d) There are three ways to obtain an instance of a class. What are they? For each alternative provide an appropriate example to illustrate your answer. **[6 marks]**

Question 2 . . . . Total: *13 marks*

The function `evens` accepts an `Array[Int]` as its formal parameter type and returns the number of even values in that array.

(a) Write a recursive version of `evens`. **[3 marks]**

(b) Write a version of `evens` that uses the `count` higher-order method. **[2 marks]**

(c) Write a version that uses the `filter` higher-order method and a regular method. **[4 marks]**

(d) Write a version that uses the `map` higher-order method and a regular method. **[4 marks]**

Question 3 . . . . Total: *7 marks*

(a) Discuss what you understand by the following statement: **[2 marks]**

"When designing an object-oriented application, a major tenet of design is *loose coupling*"

What is the role of the `new` operator in this context?

(b) How does dependency injection assist with *loose coupling*? Provide appropriate examples to illustrate your answer. **[5 marks]**

Question 4 . . . . Total: *15 marks*

Given the following classifications for some object-oriented design patterns, select two contrasting patterns from each category and provide a detailed description of each pattern. You should include in your description the problem that the pattern addresses and an appropriate example detailing the use of the pattern.

Creational Patterns Abstract Factory, Builder, Object Pool, and Singleton

Structural Patterns Adaptor, Decorator, Facade, and Proxy

Behavioural Patterns Command, Iterator, Observer, and State

Question 5 . . . . Total: *8 marks*

"The classical design patterns (mostly, the GoF patterns) are object-oriented. They show relationships and interactions between classes and objects. These patterns are less applicable in pure functional programming."

(a) Discuss this statement and provide appropriate examples to illustrate your answer. **[4 marks]**

(b) How does this statement apply to the Scala or Kotlin programming languages. You should provide an appropriate example to illustrate your answer. **[4 marks]**

Question 6 .... Total: *15 marks*

A *stack* is a data structure with a single access point (the "top") at which items are added and removed. Given the following interface representing the class `Stack`, write a class `StackFactory` that implements the *Factory Method* pattern. `StackFactory` returns an instance of `StackImplementation` which is an implementation of the `Stack` interface. Include in your answer the implementation class `StackImplementation`.

```scala
package stack

/**
 * The Stack class represents a last-in-first-out (LIFO) stack of objects.
 */
trait Stack[A] {
  /**
   * Tests if this stack is empty.
   *
   * @return if the stack is empty
   */
  def isEmpty: Boolean

  /**
   * Looks at the object at the top of this stack
   * without removing it from the stack.
   *
   * @return the object at the top of the stack
   */
  def peek: A

  /**
   * Removes the object at the top of this stack and
   * returns that object as the value of this function.
   *
   * @return object at the top of the stack
   */
  def pop: A

  /**
   * Pushes an item onto the top of this stack.
   *
   * @param item
   */
  def push(item: A): Unit
}
```

The methods should throw exceptions where appropriate.
You should provide an appropriate example to illustrate how client code should use your `StackFactory` class and deal with any exceptions that might occur.

Question 7 .... Total: *14 marks*

You are required to implement a recursive function which evaluates Fibonacci numbers and then improve its performance by caching its results with a technique called *memoization*.

(a) Write a recursive method `fib` which, given a positive integer `n`, computes and returns the `n`-th Fibonacci number. [4 marks]

```
def fib(n: Int) = ???
```

The first two Fibonacci numbers are 1. Every other Fibonacci number is the sum of the previous two Fibonacci numbers. Ensure you use recursion in your solution.

(b) Your Fibonacci method works, but it is very slow for larger numbers! If you examine the recursive calls, you should notice that some Fibonacci numbers are computed more than once. You are now required to implement a method `memo` which takes a function `f` and returns its memoized version. The memoized function stores a mutable `Map` that maps function arguments to return values. Each time the memoized function is applied to some argument it uses the `Map` to check if it was previously applied to that argument. If it was, it returns the return value associated with that argument. Otherwise, it uses the original function `f` to compute the value and returns it. [10 marks]

For example:

```
val memof = memo(fib)
memof(10)
memof(10)
```

will not recompute all the Fibonacci numbers up to 10 the second time `memof` is called.

Implement the method `memo`:

```
def memo[A, B](f: A => B): A => B = ???
```

You should use the `mutable.Map` implementation which supports the following operations (amongst others):

- `mutable.Map[A, B]` — creates a new mutable `Map` mapping keys of type `A` to values of type `B`
- `get(key: A): Option[B]` — returns `Some(v)` if the key is present in the `Map`, or `None` otherwise
- `put(key: A, value: B): Option[B]` — adds a binding from key to value into the `Map` and returns `Some(v)` if key was previously associated with some value `v`, or `None` otherwise.

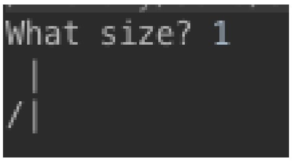
Question 8 .... Total: *15 marks*

Complete the following function:

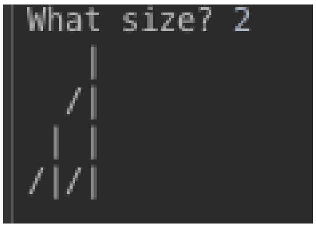
```
def createFractal(n: Int): List[String] = ???
```

The function takes an integer `n` and returns a list of strings, where the strings are rows of characters which, when printed, are the following series of fractals:

When `n==1`:



When `n==2`:



When `n==3`:

The idea is that each subsequent fractal is comprised of three copies of the previous fractal: one at the top (shifted to the right), and two on the bottom.

Hint: It will probably be useful to define *helper functions* and to use the `map` method.

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`.

Appendix B Sone Kotlin functions

`any` — Returns `true` if at least one element matches the given predicate.

`all` — Returns `true` if all the elements match the given predicate.

`count` — Returns the number of elements matching the given predicate.

`fold` — Accumulates the value starting with an initial value and applying an operation from the first to the last element in a collection.

`foldRight` — Same as `fold`, but it goes from the last element to first.

`forEach` — Performs the given operation to each element.

`forEachIndexed` — Same as `forEach`, though we also get the index of the element.

`max` — Returns the largest element or `null` if there are no elements.

`min` — Returns the smallest element or `null` if there are no elements.

`none` — Returns `true` if no elements match the given predicate.

`reduce` — Same as `fold`, but it doesnt use an initial value. It accumulates the value applying an operation from the first to the last element in a collection.

`drop` — Returns a list containing all elements except first n elements.

`dropWhile` — Returns a list containing all elements except first elements that satisfy the given predicate.

`filter` — Returns a list containing all elements matching the given predicate.

`filterNot` — Returns a list containing all elements not matching the given predicate.

`take` — Returns a list containing first n elements.

`flatMap` — Iterates over the elements creating a new collection for each one, and finally flattens all the collections into a unique list containing all the elements.

`map` — Returns a list containing the results of applying the given transform function to each element of the original collection.

`contains` — Returns `true` if the element is found in the collection.

`elementAt` — Returns an element at the given index or throws an `IndexOutOfBoundsException` if the index is out of bounds of this collection.

`first` — Returns the first element matching the given predicate.

`last` — Returns the last element matching the given predicate.

`zip` — Returns a list of pairs built from the elements of both collections with the same indexes. The list has the length of the shortest collection.

`reverse` — Returns a list with elements in reversed order.

`sort` — Returns a sorted list of all elements.

`sortBy` — Returns a list of all elements, sorted by the specified comparator.