COMP 3717 Midterm Practise

Short Answer

1) What are the main advantages of using Kotlin over Java?

2) Explain one difference between primitive and non-primitive data types in Kotlin.

3) Write a function that takes two numbers as input and returns their sum.

4) Create a simple class representing a Person object with name and age properties.

5) How would you iterate over a list of strings and print only the elements that start with the letter "A"?

6) What is a lambda expression and how can it be used in Kotlin?

7) Explain the concept of inheritance and how it is implemented in Kotlin.

8) Describe the difference between interfaces and abstract classes.

9) Given the code below, create the *eat* function

```
class Fruit(var name:String)

fun main() {

    val apple = Fruit( name: "Apple")

    apple.eat {
        println("You ate an $name")
    }
}
```

10) What are the benefits of using data classes in Kotlin?

11) Deconstruct the array below for the first two elements

```
val arr = arrayOf('a','b','c')
```

12) Create a *Pet* class that delegates the walk method using proper interface delegation reducing any boilerplate code

```
interface Walkable{
    fun walk()
}
```

13) What are the advantages, if any, of the previous *Pet* class compared to this one

```
class Dog:Walkable {
    override fun walk() {}
}

class Pet(private val dog: Dog){
    fun walk() = dog.walk()
}
```

14) Refactor the code within the red box using two different scope functions

```
val bill = Bee()
val bree = Bee()

bree.buzz()
bill.nest.add("pollen")
bree.nest = bill.nest
```

15) What is the difference between the STARTED state and CREATED state in the Activity lifecycle?

See next page for answers

Answers

1) What are the main advantages of using Kotlin over Java?

Null safety, first class functions, more concise syntax, data classes to name a few.

2) Explain one difference between primitive and non-primitive data types in Kotlin.

Primitive data types (int, double, char, boolean, float, etc) store their values directly in memory. Non-primitive types (arrays, lists, and custom objects) store addresses to their memory locations. With an exception to strings which use string pool memory.

3) Write a function that takes two numbers as input and returns their sum.

```
fun sum (a: Int, b: Int): Int {
   return a + b
}
```

4) Create a simple class representing a Person object with name and age properties.

```
class Person(val name: String, val age: Int)
```

5) How would you iterate over a list of strings and print only the elements that start with the letter "A"?

```
myList.forEach { str ->
    if (str.isNotEmpty() && (str[0] == 'A')) {
        println(str)
    }
}
```

6) What is a lambda expression and how can it be used in Kotlin?

A lambda expression, also called a function literal is a way to express a function so it can be assigned to variables, passed around as a function parameter or returned from a function

7) Explain the concept of inheritance and how it is implemented in Kotlin.

Inheritance is used when a class wants to inherit the properties and functionality of another class. By default, all classes are final, so we need to use the open modifier on the parent class to inherit it.

8) Describe the difference between interfaces and abstract classes.

Abstract classes and interfaces both provide a way to create abstraction in our code. The main differences between the two is that you can only inherit from one abstract class. They use a constructor and init. In contrast, you can implement from multiple interfaces, and they do not have a constructor.

9) Given the code below, create the *eat* function

```
class Fruit(var name:String)

fun main() {

    val apple = Fruit( name: "Apple")

    apple.eat {
        println("You ate an $name")
    }
}
```

```
fun Fruit.eat(action:Fruit.()->Unit){
        action()
}
```

10) What are the benefits of using data classes in Kotlin?

Data classes are useful when working with data. They essentially override *toString()*, *equals()*, and *hashCode()* so it's easier to manage and work with our data. The data class removes a lot of boilerplate code needed to do the same thing in java.

11) Deconstruct the array below for the first two elements

```
val arr = arrayOf('a','b','c')
```

val(a,b) = arr

12) Create a *Pet* class that delegates the walk method using proper interface delegation reducing any boilerplate code

```
interface Walkable{
    fun walk()
}
```

class Pet(walkable: Walkable):Walkable by walkable

13) What are the advantages, if any, of the previous *Pet* class compared to this one

```
class Dog:Walkable {
    override fun walk() {}
}

class Pet(private val dog: Dog){
    fun walk() = dog.walk()
}
```

With this Pet class we are limited to using only a Dog object. The previous Pet class is taking advantage of abstraction, by using an interface. It can take any object that implements Walkable creating more scalable code

14) Refactor the code within the red box using two different scope functions

```
val bill = Bee()
val bree = Bee()

bree.buzz()
bill.nest.add("pollen")
bree.nest = bill.nest
```

Example 1:

```
with(bree){
    buzz()
    nest = bill.nest.also {
      it.add(“pollen”)
    }
}
```

Example 2:

```
bree.let{
    it.buzz()
    it.nest = bill.nest.apply {
        add(“pollen”)
    }
}
```

15) What is the difference between the STARTED state and CREATED state in the Activity lifecycle?

The CREATED state is called only once before the STARTED state in the activity lifecycle. The started state might be called multiple times.