OOSD Exam

Theory Answers

11th August 2020

Course

BSc (HONS) Computer Science

Module

OO Systems Development II UFCFB6-30-2

Lecturer

Ibrahim Shahidh

Institute

Villa College

Author

Mariyam Yasmeen - S1800367

Contents

Question 2A

Explain the importance of multithreading in computer programming using Java

While most of the apps that we create for our classes deal with a single task at a time, in real life situations programs need to be complex enough to handle multiple tasks being carried out simultaneously, usually by multiple users as well. Multithreading, also called thread-based multitasking, allows us to execute multiple tasks simultaneously so that the CPU is used efficiently and its idling is minimized as much as possible as it handles these various tasks. Generally, the CPU would divide its execution time between threads and these threads can stay in a new, runnable, blocked, waiting, timed_waiting or terminated state. When a process is run, a thread is the smallest unit of execution within it.

Using multithreading when programming an application ensures that the finished product is able to use a single or multiple CPUs in the most efficient way possible and so the user gets to use a program that is able to handle multiple tasks without slowing down. Multithreaded programs provide users with a much better user experience and coders are also able to design a simpler system.

Java was designed with multithreading built in which supports creation and management of threads in its standard libraries (Gravvanis et. al, 2008). We can create threads either by the extension of the thread class or by the implementation of the runnable interface. Since Java does not have multiple inheritance support we cannot extend other classes after a thread class is extended and so in those situations where it might be required to extend another class it would be best to use a runnable interface. However, extending the thread class allows us to use inbuilt functions such as interrupt() or yield().

Question 2B

Identify and explain a situation where multithreading can be applied. Include Java code to support your answer

In a real life scenario we may have a user that needs to fetch data from a server, write data to the database and analyze the data for predicting future events etc. In this scenario if multithreading is not implemented, the user could experience hanging as the server fetches or saves data. We have to also consider the fact that in a real life scenario we would have multiple users doing multiple tasks all at once and if you do not have multithreading the server would have to listen to these requests and act upon them one by one which would which would end up wasting the user's time and CPU resources and making the whole experience terrible for the user.

If the server ran on a multithreaded environment, it would be able to concurrently attend to these tasks and use its resources in the most efficient way so as to streamline the process execution.

Another example would be in gaming where the different sprites within the game would need multiple threads so as to run the game smoothly and fluidly without the gamer experiencing any breaks in game play. This can be especially critical in games where timing is important and the program is able to identify and execute the processes efficiently.

Multithreading Example

```
class ThreadExample implements Runnable {

String threadName;

Thread t;

    ThreadExample String thread){

    threadName = threadname;

    t = new Thread(this, threadName);

    System.out.println("The new thread: " + t);

    t.start();

}

public void run() {

     try {

          for(int i = 5; i > 0; i--) {

          System.out.println(threadName + ": " + i);

           Thread.sleep(1000);

    }

    } catch (InterruptedException e) {

          System.out.println(threadName + "Interrupted");

    }

          System.out.println(threadName + " exiting.");
    }

}

class MultiThread {

public static void main(String args[]) {

     new ThreadExample("One");

     new ThreadExample("Two");

     new NewThread("Three");

    try {

          Thread.sleep(10000);

    } catch (InterruptedException e) {

           System.out.println("Main thread Interrupted");

    }

      System.out.println("Main thread exiting.");

      }

}
```

Question 3A

Why do programmers need to learn design patterns

Design patterns help a developer to analyse requirements and find object-oriented design solutions that are not tied to any languages and helps maintain a template throughout projects. By using design patterns you are able to define the architecture of the system. They provide a lot of insight due to the fact that these established solutions were based on the collective experience of developer experts. It also makes it a lot easier to debug and understand code. We have three main types of design patterns which have 23 more patterns divided among them.

Creational Design Patterns

Creational design patterns provide solutions where an object can be instantiated for certain scenarios in a suitable way. Patterns that fall under creational design include singleton pattern, factory pattern, abstract factory pattern, builder pattern and prototype pattern. For example in a singleton pattern, the pattern makes sure that the JVM has only one instance of the class.

Structural Design Pattern

Structural design patterns provide a way to set up a class structure which is useful in situations where we could create bigger objects from smaller objects with the use of composition and inheritance. Patterns that fall into structural design include adapter pattern, bridge pattern, composite pattern, decorator pattern, facade pattern, flyweight pattern and proxy pattern. For example, an adapter pattern is helpful when we need to integrate incompatible or unrelated interfaces together.

Behavioral Design Pattern

Behavioral design patterns are useful when we need objects to interact with each other. The patterns classified under behavioral design patterns include chain of responsibility pattern, command pattern, interpreter pattern, iterator pattern, mediator pattern, memento pattern, observer pattern, state pattern, strategy pattern, template pattern and visitor pattern. For example, in a chain of responsibility pattern we are able to achieve loose coupling when a client passes a request to a chain of objects.

Question 3B

One of the creational design patterns in 'Factory Pattern'. Apply and explain factory pattern in creating objects for the ordering process of a restaurant called Grill Corner where they serve grilled chicken, beef and tuna

Factory pattern, also known as factory method design pattern, is used when multiple subclasses belong to a superclass and one of those subclasses need to be returned based on an input. Using patterns through the coding process is said to improve the software quality and increases programmer productivity (Prechelt et. al, 2002)

For example we have these two subclasses that extend the food items that are served at Grill Corner

```
1   public class Menu extends Food {
2
3       private String chicken;
4       private String beef;
5       private String tuna;
6
7       public Meat(String chicken, String beef, String tuna){
8           this.chicken=chicken;
9           this.beef=beef;
10          this.tuna=tuna;
11      }
12      @Override
13      public String getChicken() {
14          return this.chicken;
15      }
16
17      @Override
18      public String getBeef() {
19          return this.beef;
20      }
21
22      @Override
23      public String getTuna() {
24          return this.tuna;
25      }
26
27  }
28
29
```

```java
public class Order extends Food {

    private String chicken;
    private String beef;
    private String tuna;

    public Meat(String chicken, String beef, String tuna){
        this.chicken=chicken;
        this.beef=beef;
        this.tuna=tuna;
    }
    @Override
    public String getChicken() {
        return this.chicken;
    }

    @Override
    public String getBeef() {
        return this.beef;
    }

    @Override
    public String getTuna() {
        return this.tuna;
    }

}
```

We are then able to define the factory class in the following way so that we can use it in the further implementation of a software which the restaurant can use.

```java
public class GrillCornerFactory {

    public static Food getFood(String type, String chicken, String beef, String tuna){
        if("Menu".equalsIgnoreCase(type)) return new Menu(chicken, beef, tuna);
        else if("Order".equalsIgnoreCase(type)) return new Order(chicken, beef, tuna);

        return null;
    }
}
```

Since factory design patterns separate the object construction from the actual code which uses the object, it helps us to make independent code that is less coupled and much easier to extend. We are also able to reuse existing objects which saves system resources. A con to factory design patterns might be that you need to implement multiple subclasses for the pattern to work.

A factory design pattern comes in handy in situations where you wish to create the objects without the client having access to the logic behind the objects that are created. In the example above, we are able to extend the superclass of food to a variety of subclasses such as the menu and order. These subclasses can decide which classes they wish to instantiate from.

Bibliography

1. Gravvanis, G.A. and Epitropou, V.N. (2008) Java Multithreading-based Parallel Approximate Arrow-type Inverses. *Concurrency and Computation - Practice and Experience* [online]. 20 (10), pp. 1151-1172. [Accessed 11 August 2020].
2. Prechelt, L, Unger-Lamprecht, B, Philippsen, M. and Tichy, W.F. (2002) Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*[online]. 28 (6), pp. 595-606. [Accessed 11 August 2020].