Turkcell Midterm

Furkan Ekici
`Turkcell Camp`
furkanekici1997@gmail.com

March 23, 2022

Contents

Abstract

This report includes the answers to the questions given in the Turkcell camp. These questions are about the structures used in Java, the differences between them, annotations and spring. All these questions are explained in detail thanks to explanations, tables, images and codes.

1 What is dependency injection ? Please give one way on dependency injection ?

1.1 Dependency Injection

Dependency injection is a design pattern that is the implementation of the "Dependency Inversion" principle, which corresponds to the letter D of the SOLID principles. [6] With this method, classes are used within each other. This method is usually applied when a class will use a method or property of another class.

1.2 Dependency Injections Types and Implementations

There are 3 types of dependency injection.

- **Constructor Injection:** The class to be used is a property of the current class. It is given as a parameter to the constructor of the current class and the assignment is made in the constructor. When an current class is created somewhere, its constructor must be given an object of the Injected class.

```
class CurrentClass{
    private InjectedClass object;
    public CurrentClass(InjectedClass object){
        this.object = object;
    }
}
```

- **Setter Injection:** It is quite similar to constructor injection. But in this way, method is used instead of constructor. The advantage of this way is that while the object given in the constructor injection is assigned once, another object can be injected at the desired location of the code in the setter injection method.

```
class CurrentClass{
    private InjectedClass object;
    public void setInjected(InjectedClass object){
        this.object = object;
    }
}
```

- **Interface Injection:** In this method, polymorphism comes to the fore. An interface is now injected, not a class. Interfaces can hold addresses of classes that implement them. Using this property, an interface is no longer a class but an interface as a property of the current class. Thus, the desired class among more than one class that implements the interface can be sent to the current class. This method is often used with constructor injection or setter injection.

```
interface InjectedInterface{

}
class InjectedClass1 implements InjectedInterface{

}
class InjectedClass2 implements InjectedInterface{

}

class CurrentClass{
    private InjectedInterface object;
    public CurrentClass(InjectedInterface object){
        this.object = object;
    }
}
```

Its usage is as follows:

```
class Main{
    public static void main(String[] args){
        CurrentClass current = new CurrentClass(new
            InjectedClass1()); // or InjectedClass2
    }
}
```

2 What is bean scope ? Would you give one sample bean scope ?

In order to better understand this subject, a few definitions are needed first. Without these definitions, it can be difficult to understand and explain the subject.

2.1 IoC Container(Inversion of Control)

IoC is also known as dependency injection.
In the above question, it was mentioned how to do a dependency injection. It's easy to replace the constructor with a class to be injected in a single line as above. But this code will create questions when it is wanted to be used in hundreds of places. In the above example CurrentClass is injected into InjectedClass1. Assuming this code is in hundreds of places and CurrentClass will now use InjectedClass2, it would take quite a while to fix this problem. Instead, Java gives this job to Spring. Spring manages these dependencies from one place and with a single change the injected class can be changed. IoC container is like a box. It gives the dependencies declared to it to the requesting class.

2.2 Bean

A bean is an object that is instantiated by a Spring IoC container. These objects are kept in an xml file named ApplicationContext. `<bean></bean>` tags are used to add a new dependency to the Spring container.

```
<beans
.... some configurations ...
>
    <bean id="id_value" class="class_path">
        ... additional configurations ...
    </bean>

    <bean id="inject" class="InjectedClass1">

    </bean>
</beans>
```

ApplicationContext.xml

This xml file is used as follows.

```
public static void main(String[] args){
    ClassPathXmlApplicationContext context = new
        ClassPathXmlApplicationContext("ApplicationContext.xml");

    CurrentClass current = new
        CurrentClass(context.getBean("inject",
        InjectedInterface.class));
}
```

[1] Now it is sufficient to update the bean class to switch from InjectClass1 to InjectClass2.

2.3 Bean Scope

Scope value can be given as an extra property when defining a bean. According to the type of this scope value, the life cycle of the object to be produced is determined. There are many types of scopes.

- singleton
- prototype
- request
- session
- application
- websocket

Except for two of them, the others are only used with web compatible applications.

2.3.1 Singleton and Prototype Scope

The two most used scope types. A bean uses a singleton as the default scope. This means that only one bean class will be produced. The same object is given to each class that wants to use this class and operations are performed on this object. In Prototype, on the other hand, it creates a separate object and gives it to each class that wants to use the bean class.

3 What is pointcut ? What is advice ?

3.1 AOP (Aspect Oriented Programming)

AOP is an approach used to reduce the complexity of a software. Complexity is reduced thanks to the modularity of cross cutting concerns. Cross cutting concerns(As seen in Figure 1) are structures that are not dependent on any layer in the system and can be used at every layer. The most common examples are logging, security, transaction etc. The purpose of AOP is to manage them. [9]

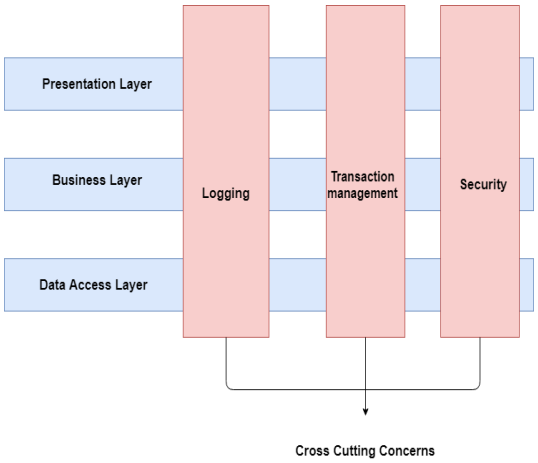

Figure 1 - Cross Cutting Concerns [3]

According to the first rule of the SOLID principles, the Single Responsibility Principle, a class or method must do a single job. For example, let there be a Manager class and an add method in this manager class. In this method, the following operations are performed:

- Transaction
- Check Business Rules
- Exception Handling
- Logging
- Request to DataAccess Layer to add

This example violates the first rule of SOLID. Here, it is only necessary to send a request for adding to the DataAccess layer and to control Business rules. However, Transaction, Exception Handling and Logging operations are also performed within the method. AOP is used to solve the mentioned problem. For example, the logging process is given to the add method with various configurations with the AOP structure, so that logging is performed without writing any logging code in the manager.

3.2 Advice

Indicates when a cross cutting concern will operate. There are several types.

- @Before : Run before the method execution.
- @After : Run after the method returns a result.
- @AfterThrowing : Run after the method throws an exception
- @AfterReturning : Run after the method is successful
- @Around : Run before the method starts and after the method ends.

In order to enable Advices in Spring projects, it is necessary to put @EnableAspectJAutoProxy annotation on the cross cutting concerns class. In Spring boot, it is not needed.
There is another important concept here. it's also a **join point** . The join point is written inside the Advice annotations and specifies which method the aspect will act on.

```
public class ExampleAspect{

    @Before("execution(public String getName())")
    public void AspectMethod(){
        //some operations
    }
}
```

This code concerns all methods whose method header is `public void getName()`. AspectMethod() is executed before running these methods. If only one class method is to be executed, the join point must be given including the package names and the class name.

3.3 Pointcut

A pointcut can be thought of as a set of join points. Multiple join points are specified using a regex-like structure. In this way, the desired methods are easily specified.

`execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)`
Beside ? It is not mandatory to enter them.
Here are a few examples to better understand this structure.

- `execution(public * *(..))` : the execution of any public method
- `execution(* set*(..))` : he execution of any method with a name beginning with "set"
- `execution( * *(*, String))` : the execution of any method with 2 parameters. First parameter can be anything, second parameter has to be String
- `execution(* com.xyz.service.*.*(..))` : the execution of any method defined in the service package

There are also alternative ways to **execution**. For example, methods at a specific location with **within** apply an aspect. With **this**, methods in response to a particular interface in bean class are aspected. Aspect is applied to methods of classes that imply a particular interface with **target**. Aspect is applied to the methods that are suitable for the parameter list given with **args**.

Pointcuts can be combined with each other.

```
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..*")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```

[7]

4 Could you explain @Repository and @Controller annotations ?

4.1 Repository Annotation

This annotation declares that database operations (insert, fetch, update, delete, etc.) will be performed in the class. It is a type of Component annotation. Component annotation is used to mark a class as a bean.

```
@Repository
public interface EntityDao extends
    JpaRepository<Entity,Integer> {
//some database oparations
}
```

4.2 Controller Annotation

This annotation is also a specialization of Component annotation. The Controller annotation indicates that a particular class serves the role of a controller. Controller annotation is used for classes in the API layer. It is generally used with RequestMapping and Responsebody annotations. Responsebody and Controller annotations are combined with RestController annotation that comes with Spring 4.0.

```
@RestController
@RequestMapping("/api/entity")
public class EntitiesController {

    @GetMapping("/endpoint")
    public void method(){
        //codes
    }
}
```

Let the program run at localhost:5000. When localhost:500/api/entity/endpoint is written, method() runs and the codes are executed.

@RestController declares that this class is a Controller. @RequestMapping is used to map web requests onto specific handler classes and/or handler methods. The @GetMapping annotation specifies an endpoint called with the get method.

5 What is the used of @Autowired ?

Thanks to the Autowired annotation, dependency injection is made implicitly. In this way, the programmer does not need to write code for dependency injection, so less code is written. Autowired can be done in different ways. These are Autowired on constructor, Autowired on property and Autowired on setter.

```
public class CurrentClass{
    @Autowired
    private InjectedInterface obj;

    //codes
}
```

Here is an example of Autowired on property. Constructor and setter are similar to each other. The method or constructor are overwritten with @Autowired.

Let InjectedClass be an interface. The bean of a concrete class that implements this is searched. Afterwards, dependency injection is done automatically. In this way, when calling CurrentClass, injection is made without the need to read the ApplicationContext file.

There is a problem here, that more than one class has implemented the InjectedInterface interface and a bean may have been created from these classes. In this case, a conflict occurs. The @Qualifier annotation is used with @Component(or subsets) annotation to fix it.

```
@Component("class1")
public class InjectedClass1 implements InjectedInterface{
}

@Component("class2")
public class InjectedClass2 implements InjectedInterface{
}

public class CurrentClass{
    @Autowired
    @Qualifer("class2")
    private InjectedInterface obj;
    //codes
 }
```

6 What is transaction ? How could u enable transaction method in spring ?

6.1 Transaction

Transaction is the process (read-write) of a user or user-program over the database or in the program. It may be that more than one interconnected operation is desired to be performed at the same time. Since the processes are interconnected, the error that may occur in the previous process will cause the subsequent processes to work incorrectly or not work at all. For example, all the data added after an add operation will be displayed to the user. If the insertion fails and data is still shown to the user, the missing data is shown.

Another example is if person A wants to send money to person B through a bank application (assuming A has enough money), assume the following transactions occur in sequence:

- A = A - 100
- SYSTEM DOWN
- ~~B = B + 100~~

Person A sent the money and account A was updated in the database. Then a malfunction occurred in the system and the system crashed. In this case, the money coming out of account A never reached account B and 100 units of money disappeared. Transaction is used to solve this problem. If a transaction is successfully terminated thanks to the transaction, all transactions are recorded in the database. If it terminates unsuccessfully, the database is reset as if that transaction had never been started. It also does this thanks to commit. If a transaction ends successfully, it commits all the transactions it has made, that is, it reflects it to the database.

6.1.1 Concurrency

More than one program or user may be working on the database at the same time. This is called concurrent. Transactions can run simultaneously if they provide 4 features(They are called ACID).

- **Atomicity :** Transactions must be atomic. That is, when a transaction starts, it must either terminate successfully or, if an error occurs, act as if it never started.

- **Consistency :** If the calculations made by the transaction are correct and the database is consistent before the transaction begins, the DB should be consistent after the transaction ends.
- **Isolation :** When multiple transactions run concurrently, they should not affect each other's value.
- **Durability :** If a transaction is committed, it should be reflected in the database.

6.1.2 Rollback

In case of system failure, the closest savepoint is returned. Log records are used to achieve this. Transactions that were active (that is, not terminated) at the time of the crash are detected and all transactions made by them are rolled back. Records of completed transactions are also determined and their changes are recorded in the database. After this step, it returns to the nearest savepoint. Thus, data consistency is ensured.

6.2 Transaction in Java

In Java, this transaction is done with @Transactional annotation. @EnableTransactionManagement is used to activate this.

For example, suppose you have a method that uploads an image. This process takes place in several stages in the method. First the image file is taken from the user, then this file is saved in a local directory and the file path is saved in the database. For example, assuming there is a problem saving to the local directory, this file may not be in the local directory but saved in the database with a null path. Another problem is that although the file has been successfully saved to the directory, there is a problem while saving it to the database. A file path not found from the database will be in the local directory. This is also inconvenient for data consistency. Thanks to the @Transactional annotation, when there is a problem with the code, all transactions are rolled back.

```
@Transactional
public void method(... file){
//save file the local directory
//
//save path to the database
}
```

6.2.1 Transactional Annotation Attributes

Some configurations can be made when using the @Transactional annotation.

- propagation : Configure to start or stop a transaction. There are many parameters.
- readOnly : Guarantee that data only reading will occur.
- rollbackFor : Unchecked exceptions are automatically rollbacked. But rollbackFor is required for checked exceptions. Takes an exception class as a parameter. If there is an error of the exception class type it receives, it performs a rollback operation.
- noRollBackFor : Unlike rollbackFor, it does not perform the rollback operation if an error of the exception class type is given as a parameter.

7 What is the difference between method overriding and overloading ?

7.1 Method Overloading

If a method has the same name but different parameter list than a method in its own class or inherited from its superclass, this event is called method overloading. So the name must be the same method signature different. The method signature is the part that contains the name of the method and the parameter list. Method header and signature can be seen in Figure 2.

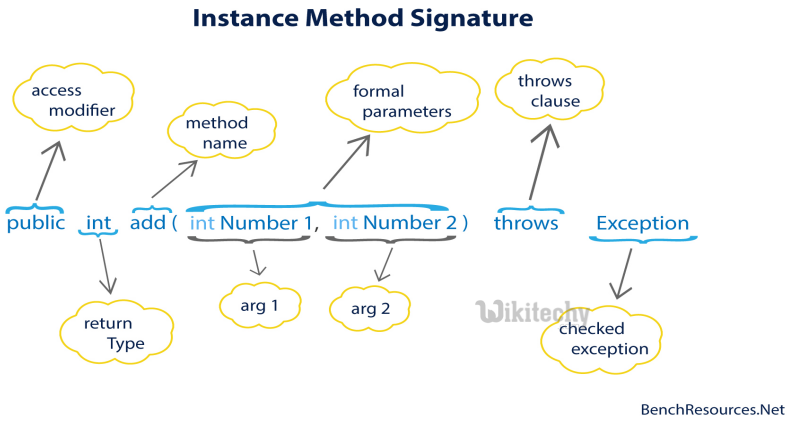

Figure 2 - Method Header and Signature [8]

```
public void add(int a, int b){
 //codes
}

protected int add(int a, int b, int c){
 //codes
}
```

For method overloading, it is sufficient that the two methods have the same name. They can have different access modifiers, different return types, different numbers and types of parameter lists, and throw different types of exceptions. Which method will work is decided in compile-time.

7.2 Method Overriding

If a method has the same method signature and return tube as a method in its superclass, it is called method overriding. Unlike method overloading, the return type must be the same here and there must be an inheritance relationship between the two classes. Two methods in the same class cannot be overrided.

```
public class SuperClass{
 public void add(int a, int b){
     //SuperClass implementation
 }
}

public class SubClass extends SuperClass{

@Override
public void add(int a, int b){
 //SuperClass implementation
 }
}
```

Private methods of SuperClass cannot be overridden. Because SubClass cannot access them. Even if the signatures and return types are the same, these are unrelated methods. Instance methods can be overridden while static methods cannot. If a static method is redefined in SubClass, the static method in SuperClass becomes hidden. The @Override annotation is used to see overridden methods and increase code readability. This use is not mandatory. When overriding a method of SuperClass, SucClass cannot downgrade its access modifier. That is, a method defined as public in SuperClass can only be public when overriding. If the method in superClass is protected, SubClass can be either protected or public. Which method will work is decided in run-time.

7.3 Overloading vs Overriding

Overloading	**Overriding**
Method overloading is used to increase the readability of code.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
Occurs with a method in a superclass or a method in the same class	Only occurs with a method in the superclass
The parameter list must be the different.	The parameter list must be the same.
Compile time polymorphism	Run time polymorphism
They can have different return type.	They cannot have different return type.
They may have different access modifiers.	They may have different access modifiers. But the access modifier of the method in the subclass cannot be less than that of the superclass.
Static methods can be overloaded.	Static methods cannot be overridden.
Private and final methods can be overloaded.	Private and final methods cannot be overridden.

8 Difference between ArrayList and vector ? Does arraylist maintains order ? Where should we use arraylist mostly?

Both of them implement List interface and use dynamically resizable array in their implementations.

ArrayList	**Vector**
ArrayList is not synchronized.	Vector is synchronized.
If its capacity is full, it will increase its own capacity by 1.5 times.	If its capacity is full, it will increase its own capacity by 2 times.
ArrayList is not a legacy class. It is introduced in JDK 1.2.	Vector is a legacy class.
ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronous, i.e. in a multi-threaded environment, it keeps other threads runnable or unexecutable until the current thread releases the object's lock.
ArrayList uses the Iterator interface to traverse the elements.	A Vector can use the Iterator interface or Enumeration interface to traverse the elements.

Vectors are synchronous, meaning that only one thread can access the Vector at a time in a multi-thread operation. Other threads wait for the previous thread to finish in order to access Vector. ArrayList, on the other hand, is not synchronized, so more than one thread can access and perform different operations at the same time.

ArrayList is fast because all threads access at the same time and there is no waiting. In Vector, on the other hand, vector works slowly as there will be a wait.

Programmers often use ArrayList because it is faster.[5]

9 What is linkedlist ? Where should we use linkedlist over arraylist ?

Actually, a linked list is a data structure. They are represented by nodes. It contains data and address parts. While the data section contains information, the address section contains the address(es) of the other node(s). There are different types of linked lists.

- **Singly Linked List :** It has a data field and an address field. The address field points the next node. The address field of the last node points the Null value.
- **Doubly Linked List :** It has a data field and two address fields. One of the address fields points the previous node and the other the next node. The head node points Null as the previous node. Likewise, the last node points Null as the next node. Java uses this type in its LinkedList structure. Its structure is as in Figure 3.

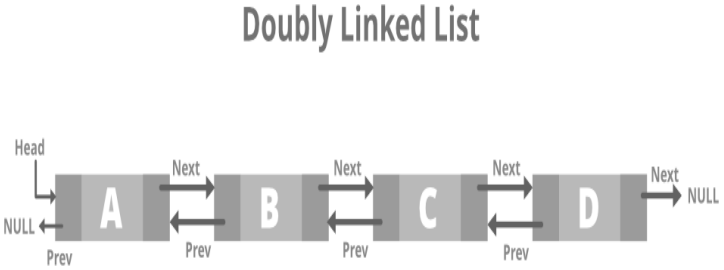

Figure 3 - Doubly Linked List [4]

- **Circular Linked List :** It has a data and address space. The last node points the head node instead of pointing Null.
- **Doubly Circular Linked List :** It has one data and two address spaces. It is a mix of Doubly and Circular Linked List.

Linked list manipulates data faster than arraylist. For example, when an element is to be deleted, ArrayList finds this element, deletes it and shifts the next elements from the deleted element in memory. This process takes a long time. LinkedList, on the other hand, finds the data and performs the deletion by only changing the address. In other words, the use of LinkedList should be preferred in cases where the addition / subtraction process is excessive.

10 Differences between ArrayList , HashSet, TreeSet ?

10.1 ArrayList

ArrayList class can be thought of as an advanced version of classical arrays. Unlike conventional arrays, ArrayLists can dynamically change size. The ArrayList in java.util is a generic class. It manages an Object Array in it. Object is a super class of all classes. [2]

10.2 HashSet

As the name suggests, HashSet is a set. There are no repeating elements in the set. Data are not retained as they are added, but instead are added according to their hash codes. Null elements can be kept in it. It is not synchronized. It is an ideal data structure for searching. Only its first and last elements can be accessed directly. Iterator is used to access others.

```
public void method(){
    HashSet<Integer> set = new HashSet<>();
    set.add(3);
    set.add(1);
    set.add(2);
    set.add(1);
    System.out.println(set.toString());

    //OUTPUT : [1,2,3]
}
```

10.3 TreeSet

It is very similar to HashSet. Different elements are sorted in ascending order, not in hash order. It is a tree implementation. It has a balanced structure. That is, instead of expanding to one side, the tree expands equally to both sides. Thus, the depth of the tree does not increase. It offers fast and easy search in huge amounts of data.

10.4 ArrayList vs HashSet vs TreeSet

ArrayList	**HashSet**	**TreeSet**
Implements List	Implements Set	Implements Set
Manages an array	Manages a hash table	Manages a tree
Allows duplicate values	Does not allow duplicate values	Does not allow duplicate values
Sort in order of insertion	Does not maintain any order	Sort from smallest to largest
It can directly access any element.	There is only direct access to the first and last elements.	There is only direct access to the first and last elements.
It can take more than one Null value.	It can take only one Null value	It does not take Null value.
It is synchronized.	It is not synchronized.	It is not synchronized.

References

[1] Engin Demiroğ. *JAVA ile Clean Code Ders 3 : Spring ile İlk IoC Yapılandırmasını Yapalım*. URL: https://www.youtube.com/watch?v=BfLhsgR_fNM&list=PLqG356ExoxZUaasgEFQDu2LAOHfVqi-w9&index=4. (accessed: 18.03.2022).

[2] Furkan Ekici. "JAVA ArrayList Class". In: (2022), p. 3.

[3] Ramesh Fadatare. *Spring AOP Tutorial for Beginners - Step by Step with Example*. URL: https://www.javaguides.net/2019/05/understanding-spring-aop-concepts-and-terminology-with-example.html. (accessed: 19.03.2022).

[4] geeksforgeeks.org. *Types of Linked List*. URL: https://www.geeksforgeeks.org/types-of-linked-list/. (accessed: 23.03.2022).

[5] geeksforgeeks.org. *Vector vs ArrayList in Java*. URL: https://www.geeksforgeeks.org/vector-vs-arraylist-java/. (accessed: 22.03.2022).

[6] Hasa Küçük. *Dependency Injection Nedir?* URL: https://medium.com/@hasann.kucuk/dependency-injection-nedir-61d2626e1103#:~:text=Dependency%20Injection%20tekni%C4%9Finde%20ba%C4%9F%C4%B1ml%C4%B1l%C4%B1k%20olu%C5%9Fturacak,s%C4%B1n%C4%B1flar%C4%B1%20d%C4%B1%C5%9Far%C4%B1dan%20enjekte%20etmeye%20dayan%C4%B1r.. (accessed: 18.03.2022).

[7] spring.io. *Chapter 6. Aspect Oriented Programming with Spring*. URL: https://docs.spring.io/spring-framework/docs/2.0.x/reference/aop.html. (accessed: 20.03.2022).

[8] wikitechy.com. *Java Method Signature*. URL: https://www.wikitechy.com/tutorials/java/java-method-signature. (accessed: 21.03.2022).

[9] Yusuf Yılmaz. *Aspect Oriented Programming*. URL: https://devnot.com/2020/aspect-oriented-programming/. (accessed: 19.03.2022).