Note: Please answer on a separate sheet of blank paper, **except for questions 1, 2, 7, 8 and 12.**

1. [4 points] – Write "true" or "false" next to statements below regarding "**Singleton**" vs. "**Prototype**" beans in Spring?
   - Prototype beans are by default eagerly loaded
   - Singleton beans are by default eagerly loaded
   - Prototype beans do not have a "destroy" cycle
   - Singleton beans do not have a "destroy" cycle

2. [4 points] - What is the order of events when Spring Context loads? Put a number next to each stage.
   - Spring sets all the properties defined for all beans (dependency injection)
   - Spring starts reading XML config file (or runs Java Config)
   - Spring calls the init() (or @PostConstruct) methods of all beans that have such a method
   - Spring instantiates all singleton beans ("Constructor Arguments" will be set at this stage)

3. [2 points] - What does Spring code below do?

```
@Value("${systemProperties.default-language}" )
private String language;
```

4. [5 points] – Consider code below. Is this "**Autowired**" by name or type? Do you see any issues? Explain.

```
@Service("customerService")
public class CustomerService {

      @Autowired
      @Qualifier("emailServices")
      private EmailService emailService;

      public void addCustomer(){
           emailService.sendEmail();
      }
 }

 @Service("emailService")
 public class EmailService { … }
```

5. [5 points] – Why would you want to use "Constructor" injection vs. field or property injection in Spring?

6. [5 points] - In the context of AOP, what is an "**Aspect**"?

7. [5 points] – Write a Pointcut Execution expression for an around advice that runs around all the **public** service layer methods. Assume that all the service layer classes are grouped in a package named: **edu.miu.cs.cs544.service**

8. [20 points] - Refactor code below and separate the cross-cutting concern into an “Around” advice.

```
@Service
public class AccountService {

 @Autowired
 private SessionFactory sessionFactory;

 @Autowired
 private AccountDAO accountDAO;

 public boolean deposit(long accountNumber, double amount) {
        Session session = null;
        Transaction tx = null;
        try {
               session = sessionFactory.openSession();
               tx = session.beginTransaction();
               System.out.println("in execution of method deposit");
               Account account = accountDAO.loadAccount(accountNumber);
               account.deposit(amount);
               accountDAO.saveAccount(account);
        } catch (HibernateException e) {
               tx.rollback();
        } finally {
               session.close();
        }
        return true;
 }
}
```

The idea is to separate the repetitive transaction/session start and stop into a reusable advice that can be applied to all the service layer methods of all the service layer classes. Now, fill in the blank below:

```
Hint: Use the Pointcut Execution expression you wrote for last question
@Aspect
@Component
public class TransactionAdvice {



 public Object transactional(ProceedingJoinPoint call) throws Throwable {










 }
}
```

9. [10 points] - Explain the "**Un-Repeatable Read**" issue and mention what isolation level solves it. How?

10. [20 points] - Explain the Spring MVC lifecycle using the following example. If you type the following in a browser: http://www.myserver.com/country-api/countries/24/cities, what happens?

```
@RestController
public class CityController {

 @Autowired
 private CityService cityService;

 @GetMapping("/countries/{countryId}/cities")
 public List<City> getCitiesForCountryId(@PathVariable Integer countryId) {

   return cityService.getCitiesForCountryId(countryId);

 }

}
```

11. [10 points] – What is security "**authentication**"? What is security "**authorization**"?

12. [10 points] – Explain the Session per Operation anti-pattern and mention at least two issues with it.

Appendix A: Validation Annotations

Annotation	Data Types	Description
@Null	Any	Check if it's null (affects column)
@NotNull	Any	Check that it's not null
@Valid	Any non-primitive	Go into the object and validate it
@AssertFalse	Boolean	Check that it's false
@AssertTrue	Boolean	Check that it's true
@Future	Date or Calendar	Check that it's in the future
@Past	Date or Calendar	Check that it's in the past
@Size(min=,max=)	String / Collection	Check size is >= min and <= max, column length set to max
@Pattern(regex=,flag=)	String	Check that it matches the regex
@Min(value=)	Numeric types	Check that it's not lower
@Max(value=)	Numeric types	Check that it's not higher
@DecimalMin(value=,inclusive=)	Numeric types	Check that it's not lower
@DecimalMax(value=,inclusive=)	Numeric types	Check that it's not higher
@Digits(integer=,fraction=)	Numeric types	Checks if it has less digits /fractions than given
@CreditCardNumber	String	Credit Cards
@EAN	String	Barcode
@Email	String	Email address

Pointcut Execution Language: **("execution(public * *.*.*(..))")**

ProceedingJoinPoint class (from AOP library) has the following methods:

Return type	Method
java.lang.Object	**proceed**(java.lang.Object[] args)
java.lang.Object	**proceed**()
java.lang.Object[]	**getArgs**()
java.lang.Object	**getTarget**()

@Before, @Around, @After, @AfterThrowing, @AfterReturning