AOP (Aspect Oriented Programming)

Spring AOP module provides interceptors to intercept an appli. for example, when a method is executed, you can add extra functionality before or after the method execution.

aspectjrt.jar
aspectjweaver.jar
aspectj.jar
aopalliance.jar

```
p. class Account
{  p. long deposit(long depositAmount)
   {
      newAmount = existingAmount + depositAccount;
      currentAmount = newAmount;
      return currentAmount;
   }
p. long withdraw(long withdrawAmount)
{ }
```

The above code models a simple Account object that provides services for deposit & withdrawal operation in the

form of Account.deposit() & Account.withdraw() methods. Suppose say we want add some bit of security to the Account class, telling that only users with BankAdmin privilege is allowed to do the operations. With this new requirement being added, let us see the modified class structure of Account.java.

```
p. class Account{
    p. long deposit(long depositAmount)
    {
      User user = getContext().getUser();
      if(user.getRole().equals("BankAdmin"))
      {
        newAmount = existingAmount + depositAmount;
        currentAmount = newAmount;
      }
      return currentAmount;
    }
    p. long withdraw(long withdrawalAmount)
    {
      User user = getContext().getUser();
      if(user.getRole().equals("BankAdmin"))
      {
```

Assume that getContext().getUser() someway gives the current user obj. who is invoking the operation. See the modified code mandates the use of adding additional if condition before performing the requested operation. Assume that another requirement for the above Account class is to provide some kind of logging & Transaction Management facility. Now the code expands as follows,

Accounts.java

```
p. class Account {
    p. long deposit(long depositAmount)
    { logger.info("Start of deposit method");
Transaction transaction = getContext().getTransaction();
 transaction.begin();
try { User user = getContext().getUser();
  if(user.getRole().equals("BankAdmin"))
  { newAmount = existingAmount + depositAmount;
    currentAmount = newAmount;
  } transaction.commit();
  } catch (Exception e) { transaction.rollback();
} logger.info("End of deposit method");
  return currentAmount;
  }
p. long withdraw(long withdrawalAmount)
  { logger.info("Start of withdraw method");
Transaction transaction = getContext().getTransaction();
  transaction.begin();
try { User user = getContext().getUser();
  if(user.getRole().equals("BankAdmin"));
  { if(withdrawalAmount }
```

The above code has so many disadvantages. The very first this is that as soon a new requirements r coming it is forcing the methods of the logic to change a lot which is against the Software Design. Remember every piece of newly added code has to undergo the Software Development Life cycle of Development, Testing, Bug fixing, Redevelopment testing. This certainly cannot be encouraged in particularly big projects where a single line of code may have multiple dependencies b/w other components or other modules in the project.

DATE: / /
PAGE: 1

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework. Aspect-oriented programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns & these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, declarative transactions, security, caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.
Dependency Injection helps you decouple your appli objects from each other & AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such a perl, .NET, Java & others.

Spring AOP modules provides interceptors to intercept an appli. for example, when a method is executed, you can add extra functionality before or after the method execution.

AOP Terminologies →

Aspect : This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An appli can have any number of aspects depending on the requirement.

DATE: / /
PAGE:

Join point → This represents a point in your appli. where you can plug-in the AOP aspect. You can also say, it is the actual place in the Appli. where an action will be taken using Spring AOP framework.

Advice → This is actual action to be taken either before or after the method execution. this is an actual piece of code that is invoked during the prog. execution by spring AOP framework.

Pointcut → This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns.

Introduction → An introduction allows you to add new methods or attributes to the existing classes.

Targeted Object → The obj. being advised by one or more aspects. this obj. will always be a proxi -ed. obj., also referred to as the advised obj.

Weaving → weaving is the process of linking aspects with other appli. types or obj. to create an advised obj. this can be done at compile time, load time, or at runtime.

DATE: / /
PAGE:

```
package com.myStudent;

p. class MyAspect
{
/* this Method execute before select method execution.*/
p. void beforeAdvice()
{
  Sop("Going to setup student profile");
}
/* this method execute after a selected method */
p. void afterAdvice()
{
  Sop("Student profile has been setup");
}

/* this method execute when any method returns */
p. void afterReturningAdvice(Object retVal)
{
  Sop("Returning:" + retVal.toString());
}
/* this is the method will execute if there is an
   exception raised */
p. void AfterThrowingAdvice(IllegalArgumentException
                            ex)
{ Sop("There has been an exception" + ex.toString());
}
}
```

DATE: / /
PAGE:

```
package com.myStudent;

p. class Student
  { private Integer age;
    "      String name;
p. void setAge(Integer age)
  { this.age = age; }
p. void getAge()
  { Sop("Age"+age);
  }

p. void printThrowException() { Sop("Exc raised");
        throw new IllegalArgumentException(); }}
```

```
package com.mystudent;
import org.springframework.context.ApplicationContext;
                .support.ClassPathXmlApplicationContext;
p. class RunAop
  { psvm()
    { ApplicationContext context = new ClassPathXmlApplication
                                    Context("Beans1.xml");
Student student = (Student) context.getBean("student1");
  student.getName();
    ".  getAge();
    ".  printThrowException();
  }}
```

```
package com.Lucad;
import org.aspectj.lang.annotation.Aspect;
                .Pointcut;
                .Before;
                .After;
```

- AfterThrowing;
- AfterReturning;
- Around;

```
@Aspect
p. class Logging {

@ Pointcut("execution(* com.ducat.*.*(..))")

private void selectAll()
  {}
@Before("selectAll()")

p. void beforeAdvice()
  {
    Sop("Going to retrieve student profile");
  }
@ After("selectAll()")
p. void afterAdvice()
  { Sop("Student profile has been retrieved");
  }

@ AfterReturning(pointcut = "selectAll()",
                 returning = "retVal")

p. void afterReturningAdvice(Object retVal)
  {
Sop("Returning "+ retVal.toString());
  }

@ AfterThrowing(pointcut = "selectAll()",
                throwing = "ex")
```

DATE: / /
PAGE:

```
p. void AfterThrowingAdvice(IllegalArgumentException
                            ex)
{
  Sop("There has been an exception "+ ex.toString());
}
```

Student

Run Spring