**AOP - Pointcut** –

A set of rules determining to which methods, and in which classes a piece of advice should be applied.

An advice is a code that will run around functions. Aspect on the other hand is like a module in the system which is target for applying an advice. Aspects are – Transactions, logging, security.

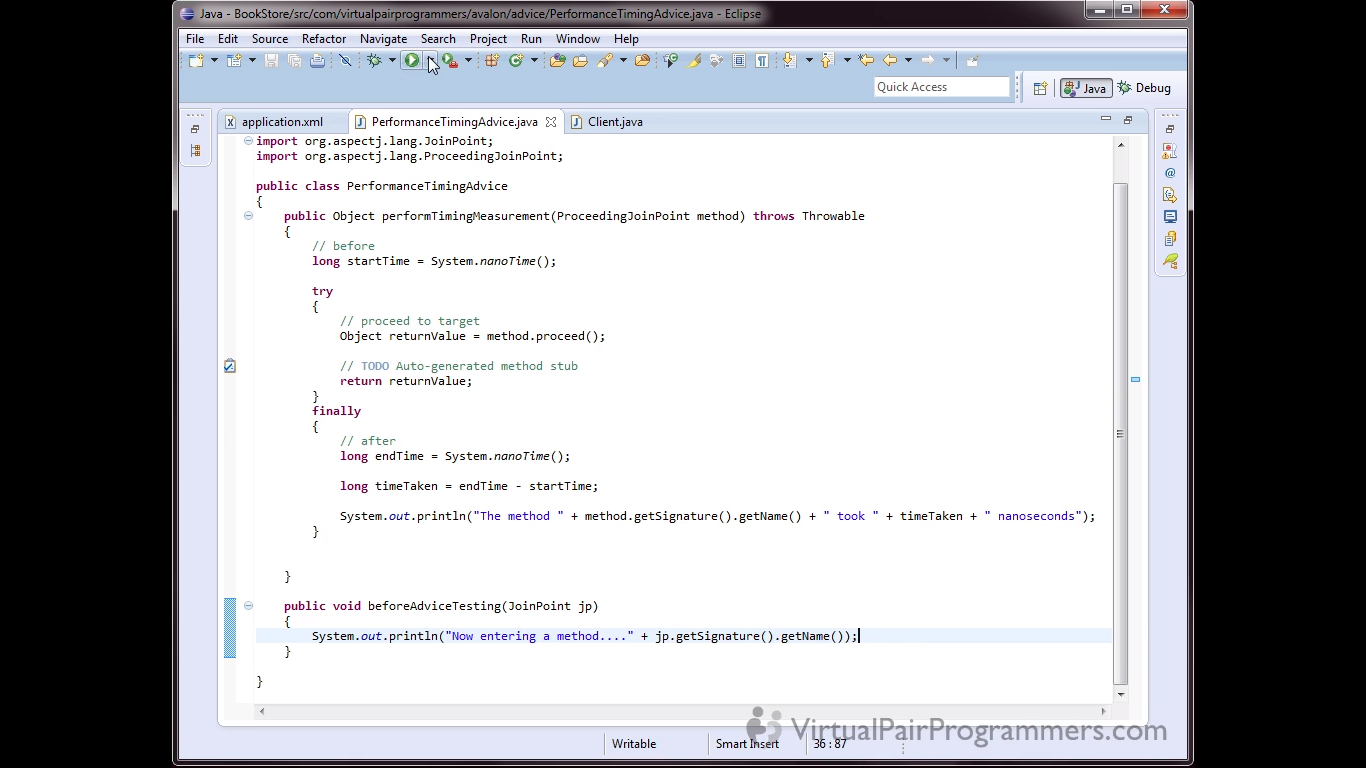
Spring has borrowed AspectJ syntax from spring 2.0 onwards.

\*Service\*.get\*(. .) = all services beginning with get with whatever number of arguments

\* \*Service\*.get\*(. .) = Above example plus all type of return types.

execution(\* \*Service\*.get\*(. .)) = All method execution in service class beginning with get .

Configuring AOP –

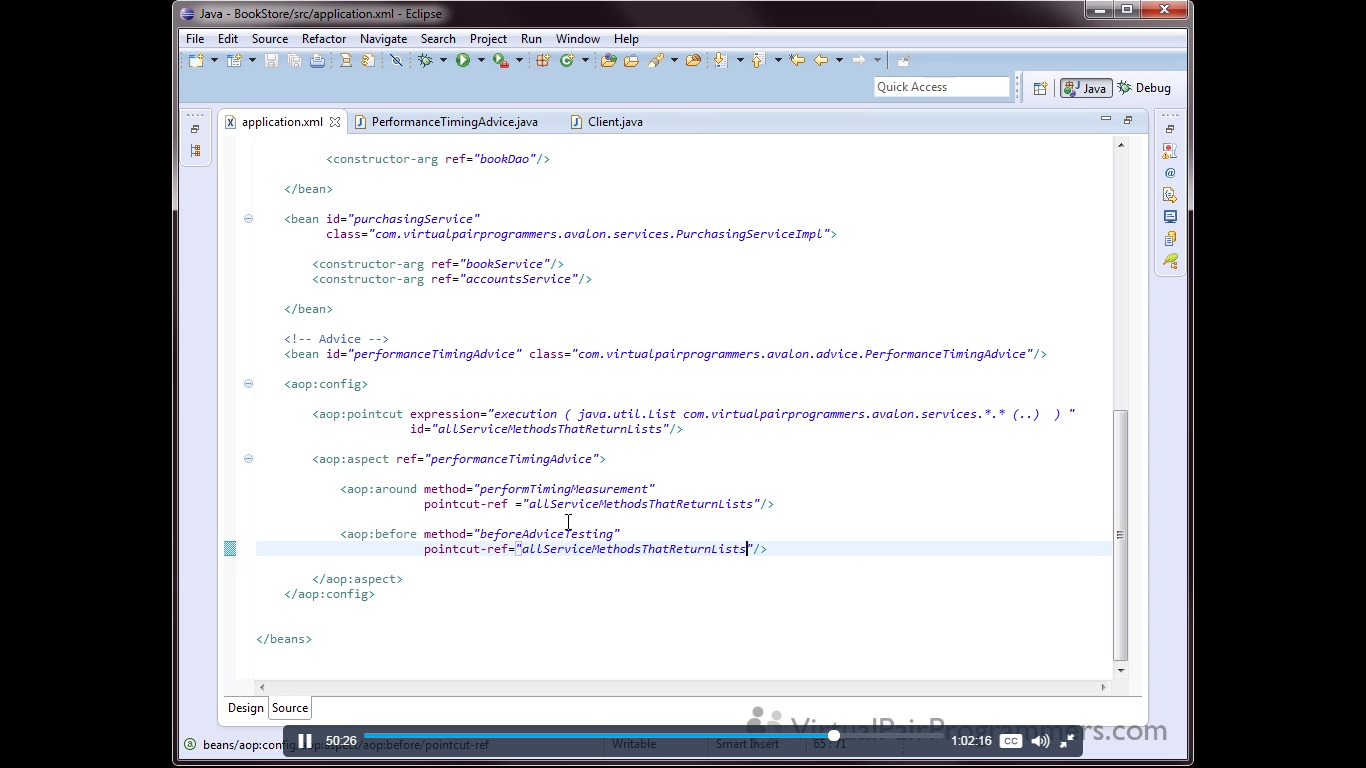


The method performTimingMeasurement is an around advice. It will take ProceesingJoinPoint object as argument as we will need method.proceed() method to call the object method and then wrap the advice code logic before and after the method call.

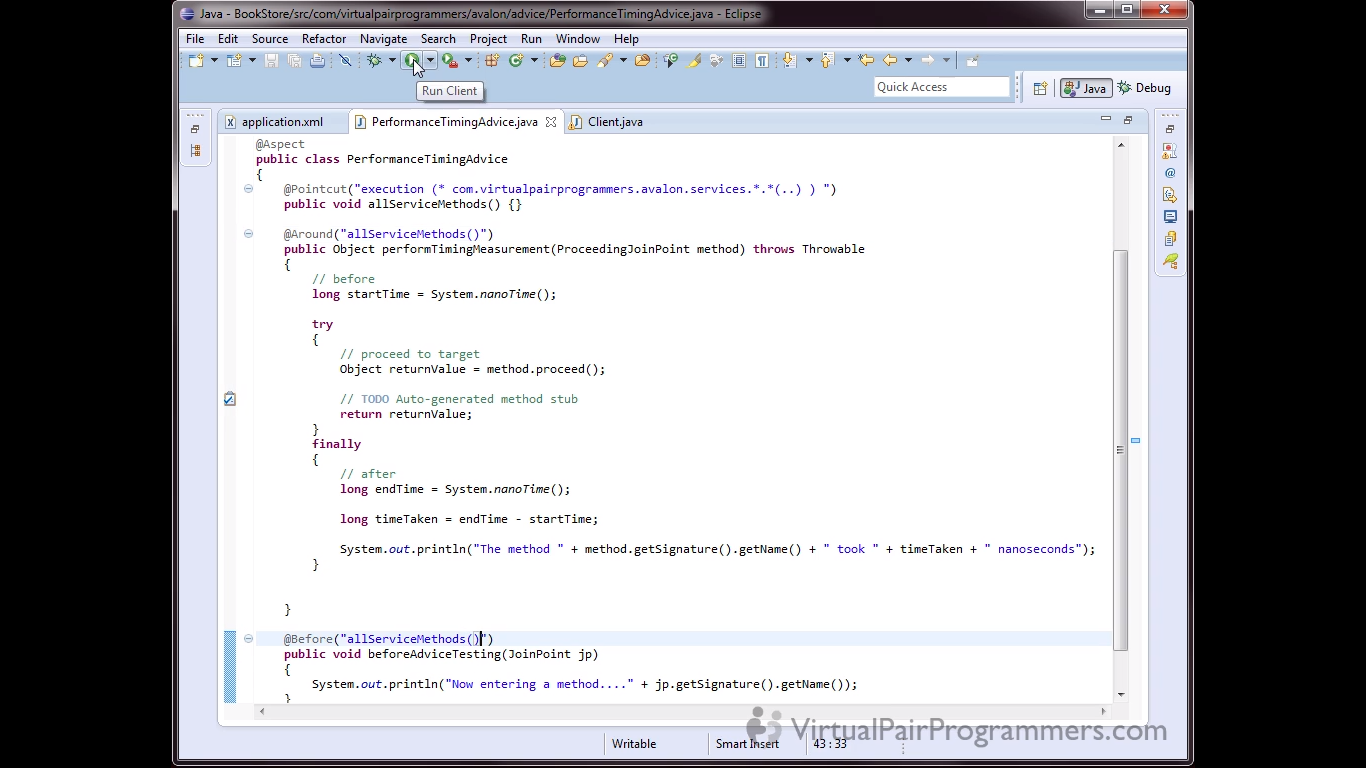
The second method is only a beforeAdvice and if we need any additional info about the method, we will simply need a JoinPoint object as method argument.

We can remove the xml configuration altogether and use only annotations. Just use the configuration

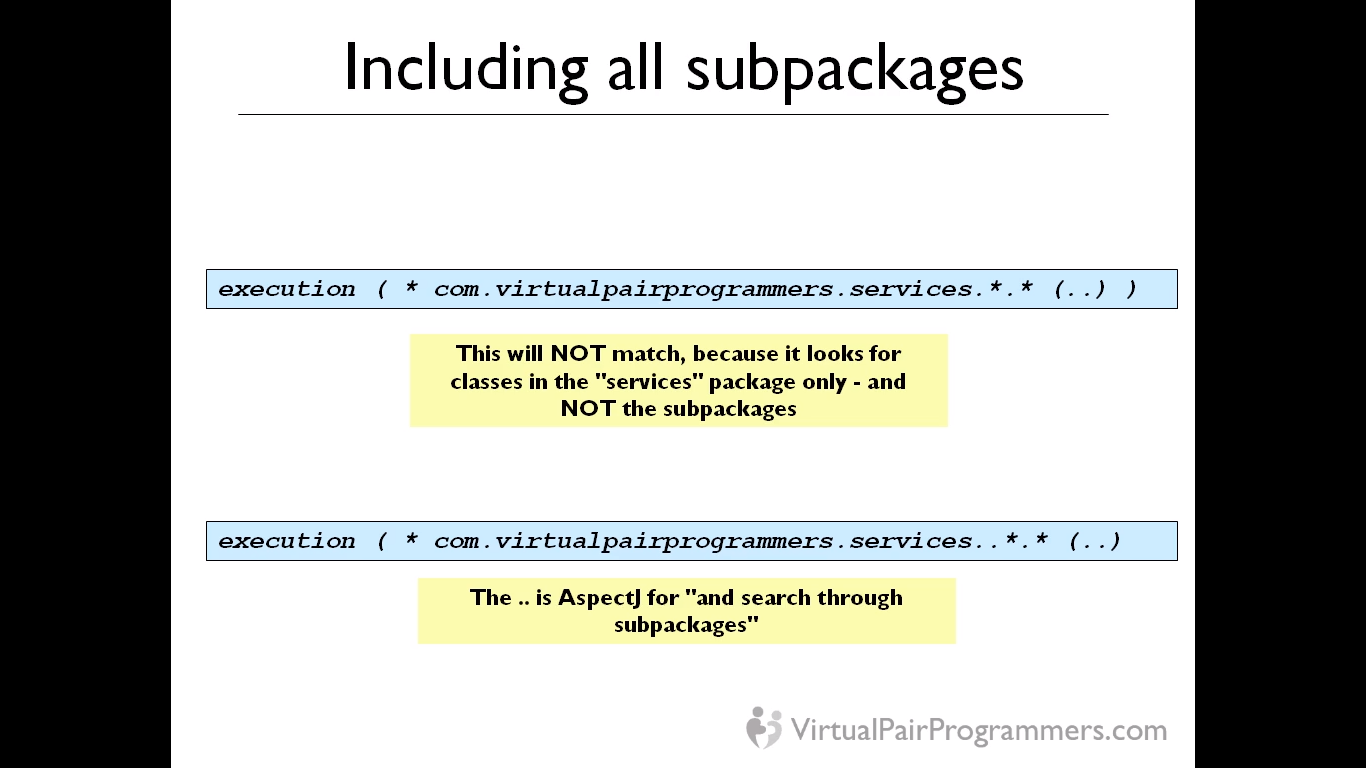
**<aop:aspectj-autoproxy />.** This will tell spring to search for aspect related annotations.



The same thing with the annotations in the class



The empty method allServiceMethods() is an hook to start the advice and logic is written over than method. Rest of the advice function will use the logic written over that empty method. Some people do prefer the xml based configurations as xml gives flexibility to change the logic of searching the methods in xml and changing than in xml will not need a recompile of code.



If we don’t put and extra dot after services, it will not include all the sub-packages. Also one weakness of AOP is that, the aspect will only be applied to those classes configured in spring container. The classes out of the container or not annotated with spring annotations, won’t get any aspects logic at all.

Pointcuts can be combined with || operator like below

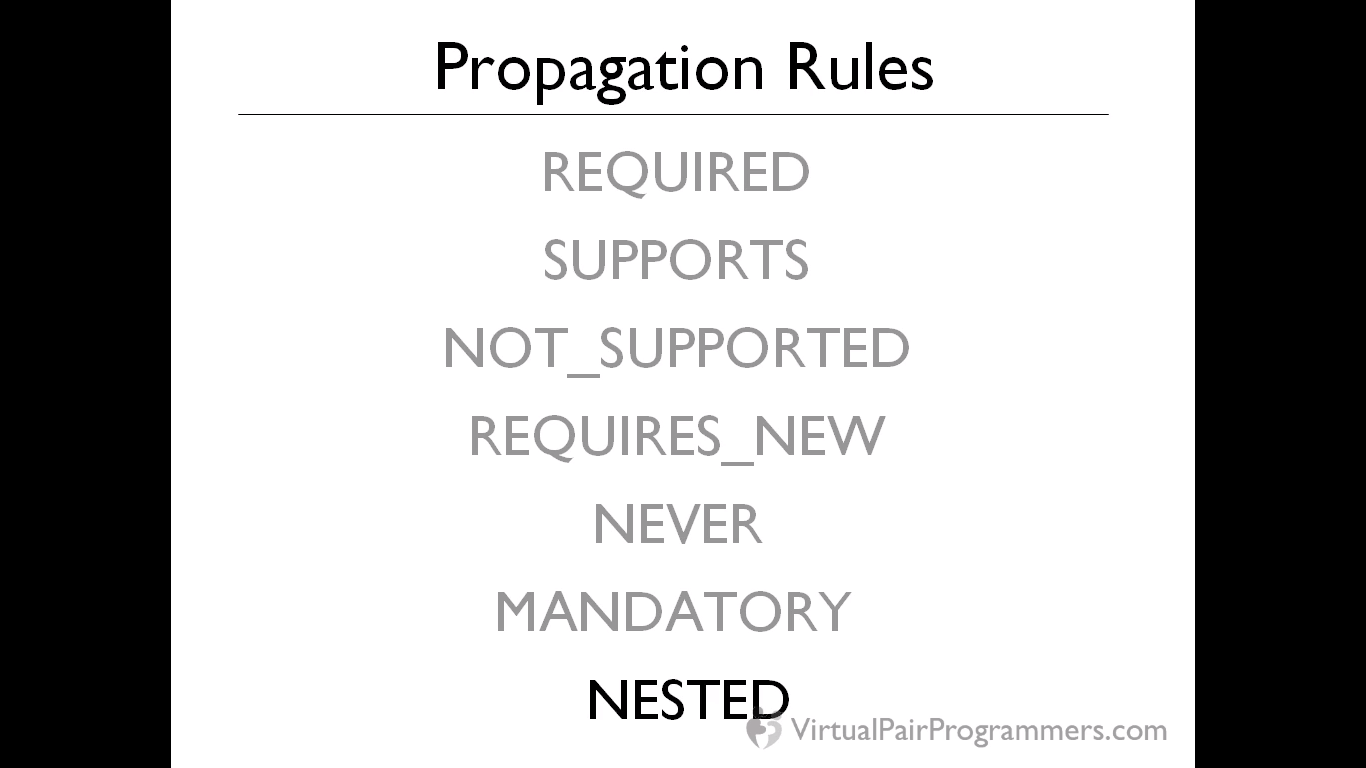
execution (pointcut1..) || execution (pointcut2..)

**Transactions**

Usually best place to put @Transactional annotation is to put it at Service level. To configure transaction, we need to define a transaction manager bean which is like a spring advice. Since the DAO object will have only single database operations while service function will have multiple function calls to DAO layer and hence a good place to have a transactions annotations.

For transactions, preferred method is to have annotations.

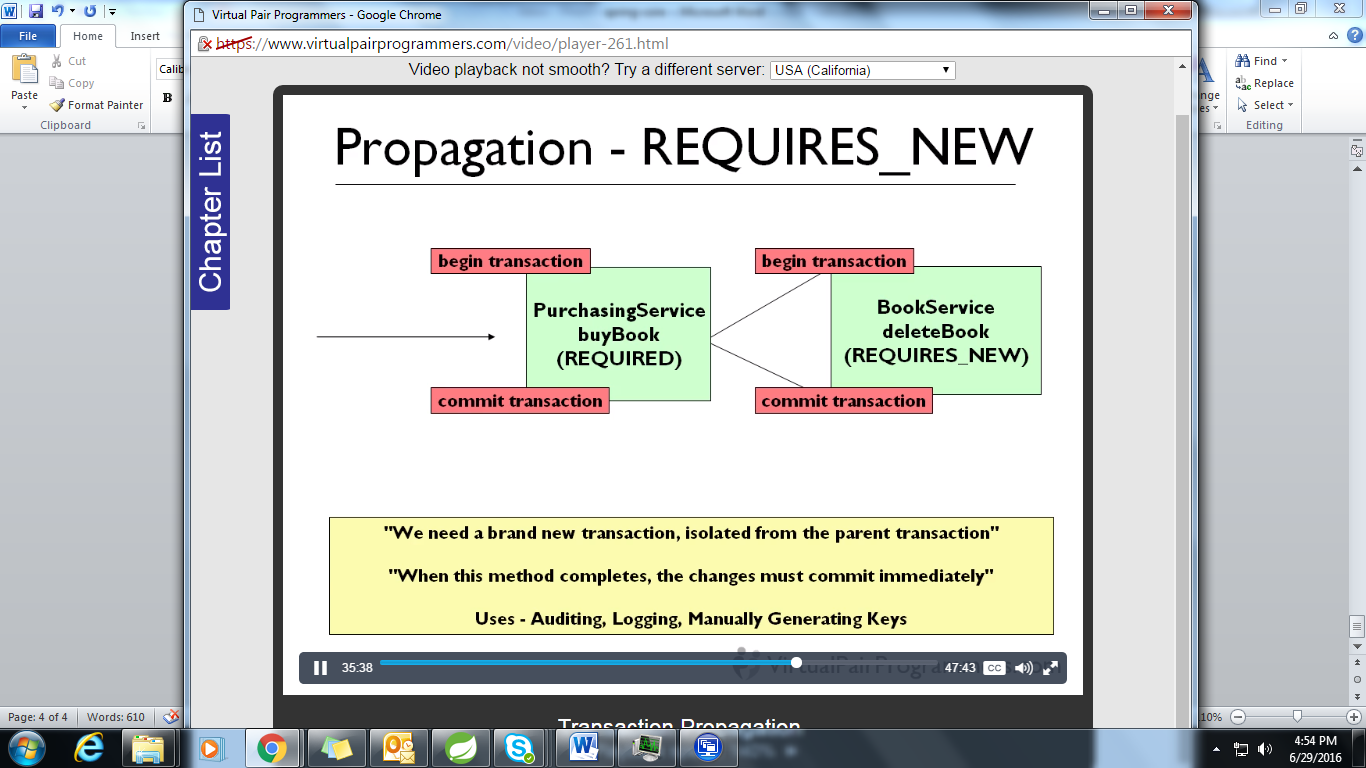
Propagation rules – Default behavior, nested function calls will join the existing transaction. No new transactions will be created.



**Required** – Nested function/child function will require a transaction but if the calling method has already initiated the transaction, then child method will join the transaction. This is the default behavior.

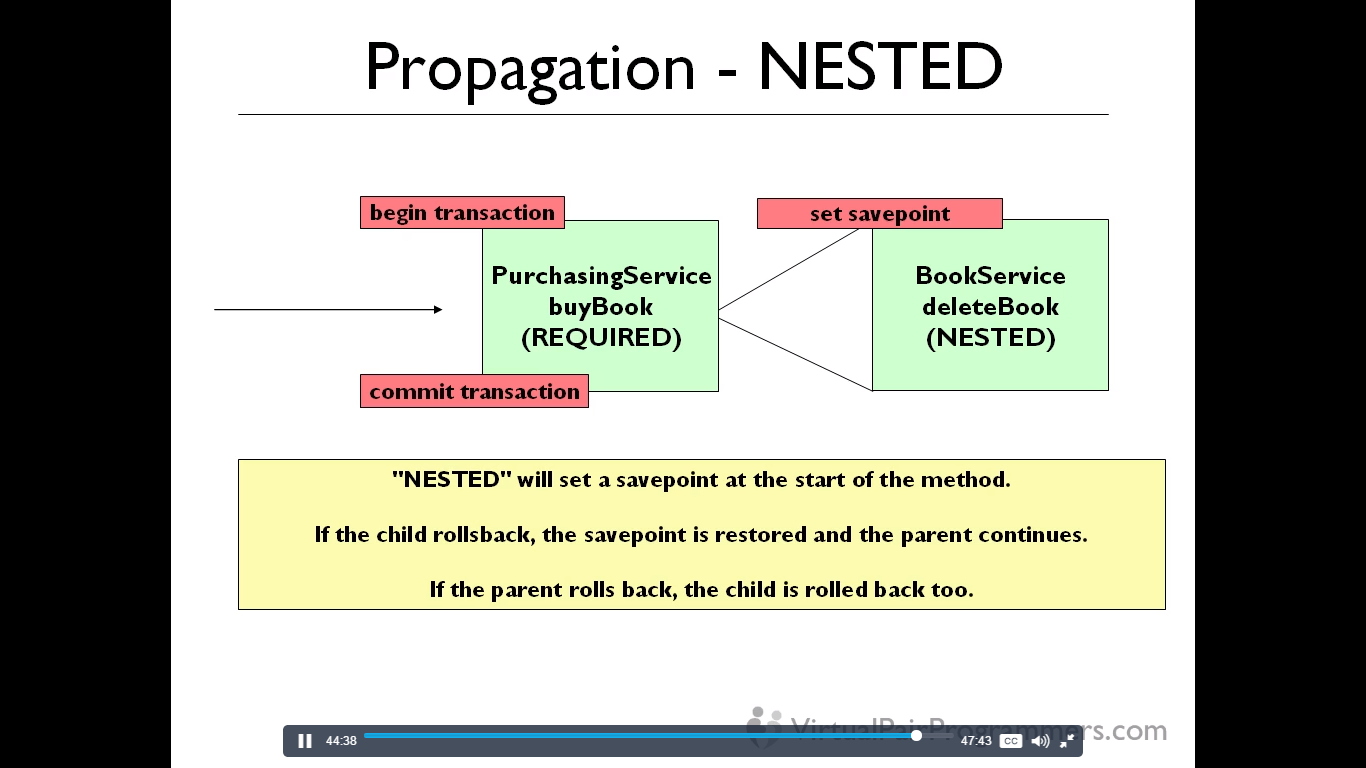
**Supports** – Don’t bother creating a new transaction, but we don’t mind if there is already one. This is useful if the nested method (may be from different service) is not using a DB call to handle. If transaction is present, the method will participate, else it doesn’t care if is not there. We should always identify methods that don’t require transactions and act accordingly.

**Not\_Supported** – The child method will suspend the existing transactions if any and will run in a different thread altogether. After the function is over, the transaction is resumed. This is used only for those methods where open transaction may affect the calling child method inside. Example – if from a child method, stored procedure is called with some create table or update table, this will result in a db crash if call is from an open transactions. “We definitely do not need transaction for this method”.



This propagation rule is rarely used as child method will start an altogether new transaction and will be committed even if the parent method will fail afterwards. So only an independent logic will be written in child method. Like auditing, this irrespective of parent method runs or fail will log the event that parent method was executed.

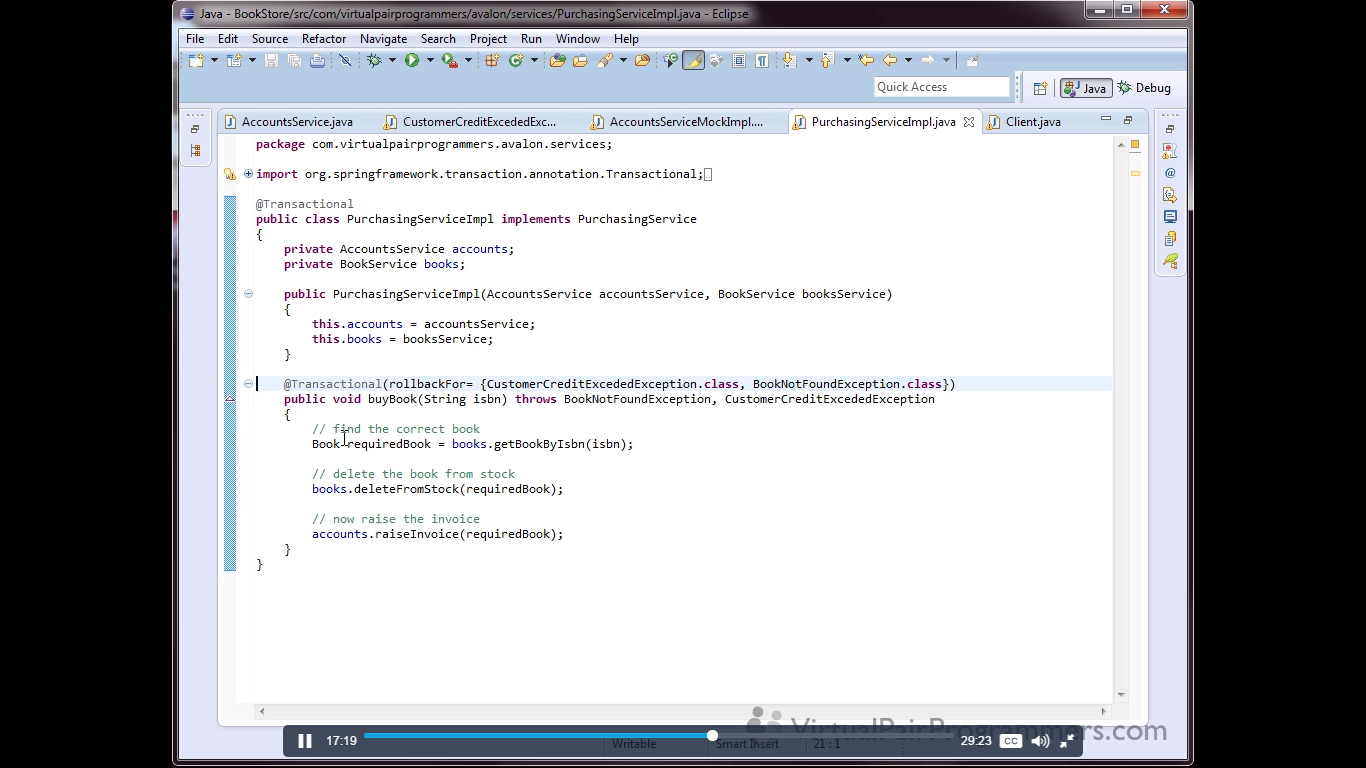
**Mandatory** – A mandatory transaction is needed in the child function. Example – when we want to force the client to use service method and not directly the DAO methods so we can write a transaction = mandatory in the DAO classed and also we already have a @Transactional annotation in the service classes. So transaction will be first initiated in the service classes only. So any direct call to the DAO classes will result in exception as no existing transaction will be found.



**Nested** – It won’t work with hibernate and distributed environment but with JDBC.

**Rollbacks**

Default in Spring is that, Checked exceptions do not result in rollbacks. Let’s say some programming mistakes and catching and throwing some custom exceptions, would not result in rollbacks and we need to manually configure in the code to issue a rollback.



We can write a method level transactional annotation and letting the spring know that you will need a rollback issue for the number of checked exceptions.

We can also put transactions for a read-only token as not putting the transactions will not save any performance.

<http://stackoverflow.com/questions/818074/transactions-for-read-only-db-access>

We can have a @Transactional(readOnly=true) over a method which supports only readonly so we can call it directly without starting a transactions. This can improve a performance as some database maintains a file with steps so that if a rollback can be initiated, it will undo the steps. So for readonly functions, db will not create a file.

We can optionally set a timeout attribute for the @Transactional annotations. Sometimes in a function call, if some method do takes more time, and user does not waits and closes the browser. So in such cases, we can provide and optional attributes to be set with timeout.