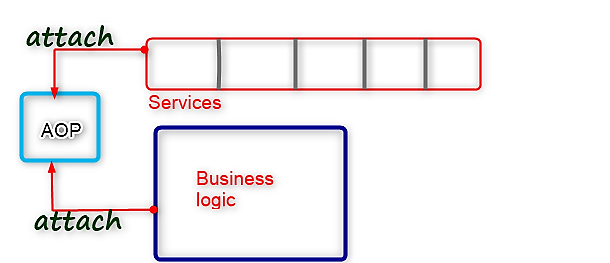
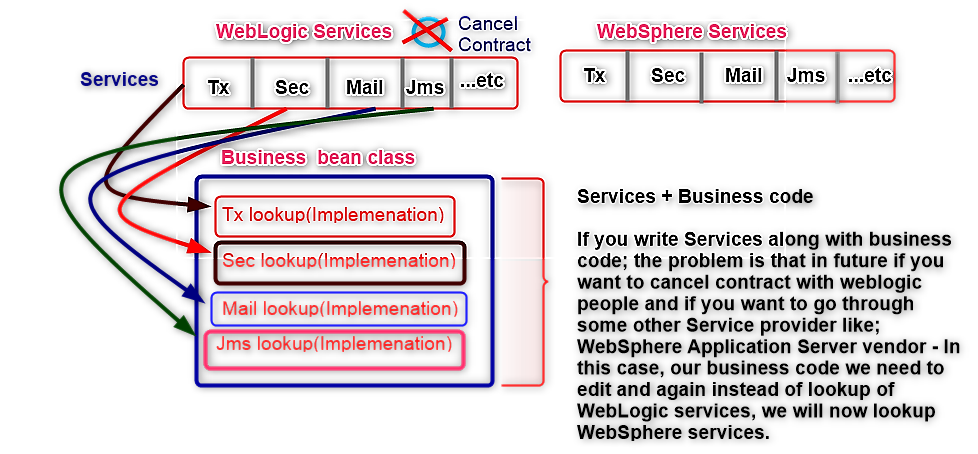
**Aspect||Service Oriented Programming.**

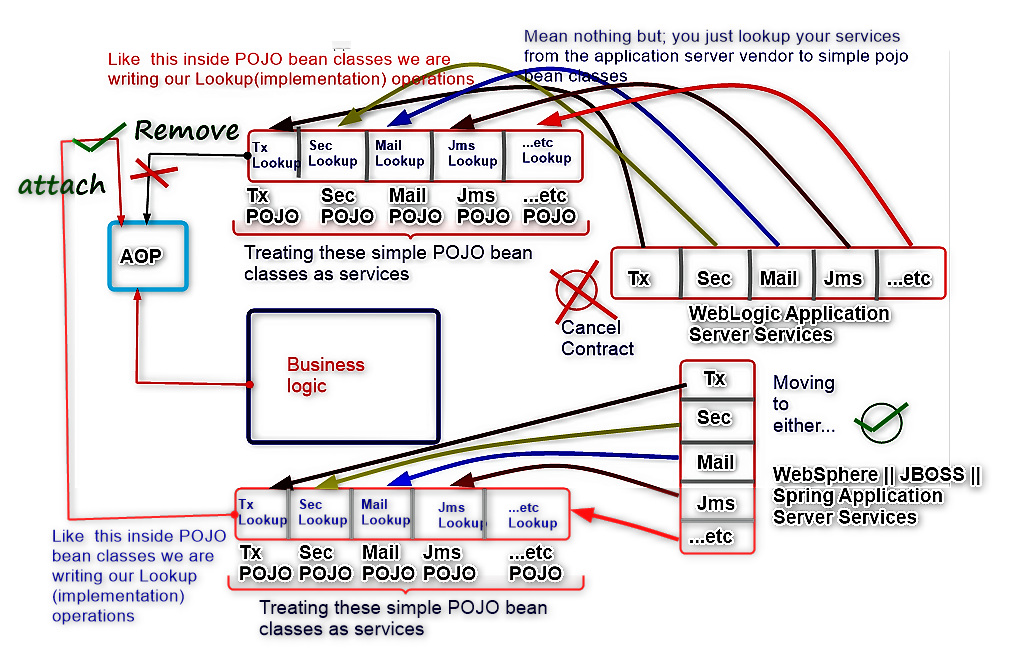
* **AIM**: Instead of writing business along with service(Combining) in one bean class; using AOP, we can write services **separately** and add that to business logic using AOP.
* Nothing but AOP concept will help us **separate** services from business.



* Generally, if you don’t have AOP; what happens?
* Whenever you require services in your business; let’s say if services you are going to **buy** from: **WebLogic**;
* **Services are usually provided by application server vendors-3rd party** or else we can use **Services provided by Spring**.
* Spring can provide **services||third-party** vendors can provide services.
* So, third-party WebLogic can provide:
* **Tx** Service
* **Security** service.
* **Mailing** Services.
* **JMS** Service.
* If these services you are going to use in your business; usually inside our business bean class(s); the respective service from respective Service vendor, i.e., WebLogic in our case, we need to **lookup(Implement)** inside our business class(s).
* Whenever you require:
* **Tx services** 🡪 That you need to **lookup**(implement) inside your business class.
* **Sec Services** 🡪 That you need to **lookup**(implement) inside your business class.
* …etc.
* See image below for clarity.



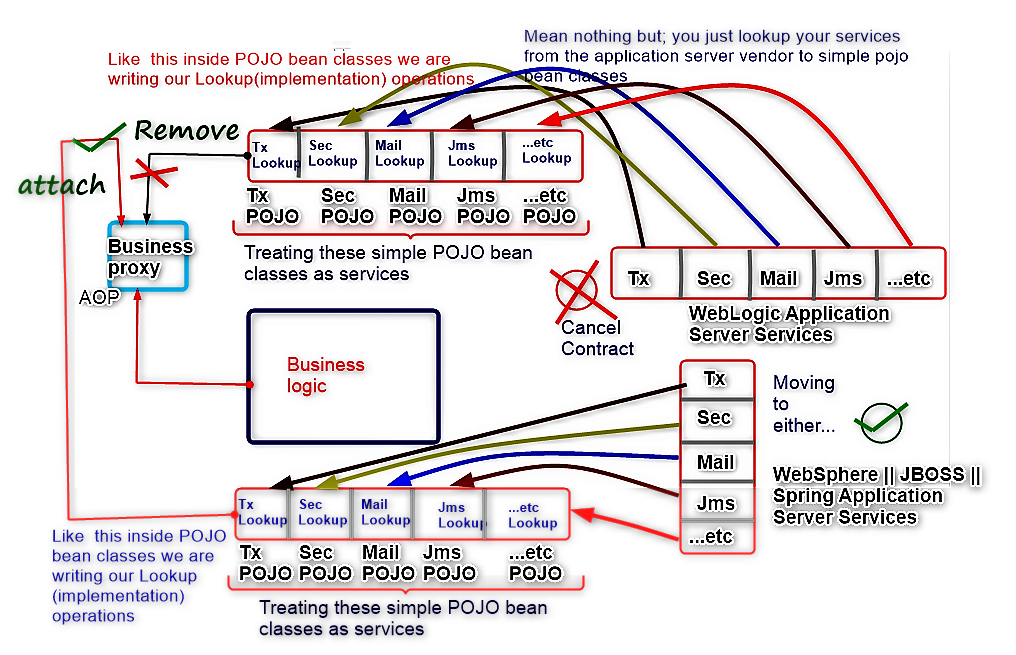
* Like this, from the image above; since we are writing WebLogic code **directly** inside our business class, in future if you want to migrate your application from WebLogic to WebSphere; your **business code you need to change||edit**.
* Another problem with this kind of code is; whenever you require changes on services , that service changes may affect your business code.
* So, instead of combining Services + Business code together like in the image above; using AOP, we can write services and business code **separately** hence avoiding the issues discussed above.
* Let’s say you want to use **WebLogic** services; In case of AOP what happens?
* That Services code you can separately write in :
* Spring Service implementations OR
* **Simple POJO classes** and then in that POJO classes **you can lookup(Implement) these services** from your required application server vendor.
* See image below for clarity:
* Then, at **runtime** you can attach your required **services dependency** of application server vendor i.e., WebSphere to your AOP and we can remove the old services dependency of application server vendor. i.e., WebLogic from your AOP.
* Now, instead of executing your services from WebLogic; now onwards, your business class will execute services from WebSphere.
* In future if you don’t want WebSphere services; and you know that Spring is providing free services -Actually Spring has it’s own implementations for services- Spring services: **Tx**, **Sec**, **Mail**, **Jms**, ..etc., if you want to use for your application: then you can go ahead and lookup(Implement) that services inside your POJO classes-that we are treating as services-[**Inside service package**], then at **runtime** you can **attach** that service implementations(Present in service package) to your AOP and you can remove your old application server services dependency from your AOP.
* Like this; your business bean class has **NO** particular application server service dependency. But if you directly lookup(Implement) service code inside your business class; then your business will become service dependent.
* This is thus the main **AIM** of AOP concept. To make Business Layer independent of Service Layer and combine the two at runtime.



* So, to maintain independence of service layer for our application, we thus need to go through AOP concept as discussed in details above.
* Using AOP, at **runtime** we can add services to our business logic||code. What does this mean?
* While executing your business code, along with business code it will execute your services code.

**How will it execute?**

* AOP internally; whenever you **attach** Services and Businesses to AOP, internally AOP will **generate proxies**.
* In Java; **Proxy** means🡪A **duplicate** instance.
* For the **Business** class, it will generate one **Businessproxy class**. Nothing but; **Businessproxy class** means that it is a **duplicate** instance of Business class.
* In this **Businessproxy**, it will add services(Tx, Sec, Mail, Jms,..etc.).
* Your services it will attach to your business code and then that combination it will attach to your Businessproxy class.
* It will in short generate a proxy class-**Businessproxy class** and that proxy class will have a combination of your **Services** and **business** code.
* Finally, AOP will provide||return us the **Businessproxy object**. This resulting object, we can store it into Business class reference-Parent class reference
* Whenever you execute; **businessproxy.methods** , usually business methods you can be able to execute. But while executing these business methods, along with your business logic||code inside the business method; the services also it will execute.
* In a nutshell, AOP job is:
* **AIM**: Ensures a **separation** between business layer from service layer(Make them independent); then finally **combining** these two separations inside a generated runtime proxy class.
* **Who will do the combining actually**?
* AOP proxy classes.
* In AOP we have some proxy classes; that proxy classes: Services code they will combine with business code.



**Terminologies.**

The below terminologies are not related to OOP.

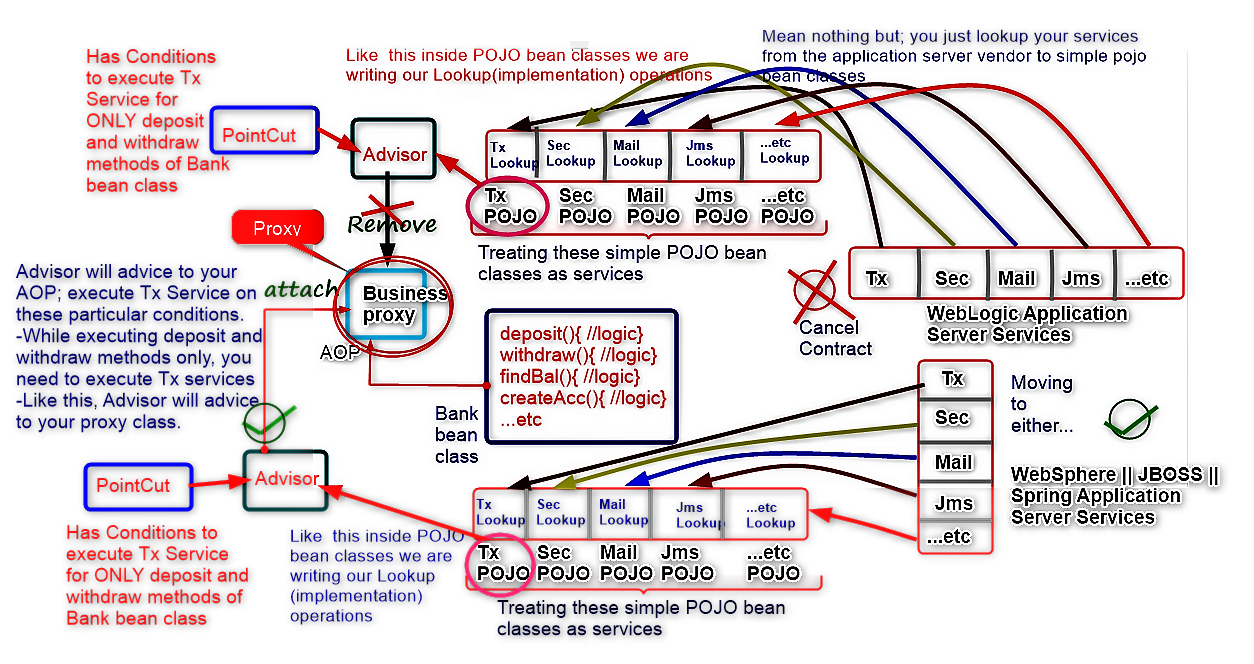
* **Aspect**: Means **Service**.
* Like this you can also say: **S**ervice **O**riented **P**rogramming.
* To add Services to our business we are using this concept.
* **Advice**: Means Aspect||Service provider.
* In the programmatic approach, For Spring Framework; we have **4** advices:

1. **MethodBeforeAdvice.**
2. **AfterReturningAdvice.**
3. **AroundAdvice||MethodInterceptor.**
4. **ThrowsAdvice.**

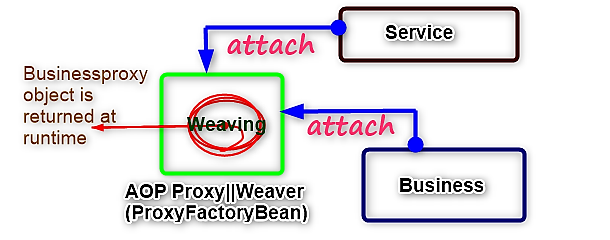
* **Pointcut**: Means; **Service||Aspect execution condition point**
* This implies that at a particular point, you may require a particular service. i.e.,
* At some point you may require a **Tx** service.
* At some point you may require a **Sec** service.
* At some point you may require a **Mail** service.
* …etc.
* If the Business class is: Bank class. We may have business methods like:
* deposit()
* withdraw()
* findBal()
* createAcc()
* calIntrest()
* delAcc()
* fundTransfer()
* …etc.
* Whenever you require to **deposit** or **withdraw**; in these two cases if you want to execute **Tx services** and for some other cases like; **findBal**, **createAcc** and **delAcc**, …etc. if you don’t want to execute your Tx service. Let’s say:
* If you are going to add **Tx service** to this Bank business class, before each and every method execution, AOP will execute all your Services. By using a **decorator pattern** actually, it will execute.
* The decorator pattern will do what? While executing your methods, along with your methods; the services code also it will execute.
* Nothing but, whenever I try to execute deposit method-Tx Service it will execute. Whenever I try to execute withdraw method-Tx Service it will execute. Whenever I execute any of the remaining methods; (findBal, createAcc, delAcc, ...etc.)-Tx Service it will execute.
* Now, whenever I execute some of the remaining methods; (findBal, createAcc, delAcc, ...etc.)-In these cases, if I don’t want to execute Tx service and I needed to execute Tx service **ONLY** for some methods, i.e., deposit and withdraw methods; then you need to **apply||pass conditions to the AOP making it understand your requirement.**
* It is through **pointcut** that you need to pass that **condition**.
* Based on those conditions, AOP will execute your services for particular methods only.

**NOTE:**

* To know balance from account, Tx’s not required. This is also true for: creating account and deleting account)
* Tx’s are required for **DML operations**.
* In case of deposit and withdraw; you may need to perform DML operations on your DB.
* In case of find balance; it is a DRL operation and thus for such operations: Tx’s not required.
* If you are dealing with multiple **Data Access Operations**, then ONLY this Tx’s will be required to your methods.
* deposit method, may interact with only one table
* withdraw method, may interact with only one table.
* fundTransfer method, may need to interact with multiple tables.
* In this case; fundTransfer method may require Tx’s. The remaining all other methods may NOT require Tx’s.
* If you write a Service for Bank class, then; that service will execute for all those bank class methods. Instead of executing for all those methods; if you want to execute for only particular methods, i.e., fundTransfer method; then you have to apply conditions by using Pointcut.
* **Advisor**: Combination of **Aspect||Service** **+** **Pointcut**.
* Nothing but the combination of: **Service** **+** **Condition**.
* Let’s say; for Tx Service we are adding a pointcut-condition.
* The Condition we are adding is; execute Tx Service only for deposit and withdraw methods
* In this **Pointcut class** if we write conditions as above: these conditions applying for;
* Tx service
* So, you need to combine this Tx service + Pointcut using: **Advisor**.
* Finally, this Advisor will advise to your AOP; execute Tx Service on these particular conditions.
* While executing deposit and withdraw methods only, you need to execute Tx services
* Like this, Advisor will advise to your AOP proxy class; **ProxyFactoryBean**
* **Advisor** job is; advise a service on a particular condition
* See image below for clarity.



* **Proxy||Weaver**: Will add Services code to Business code and finally generate a proxy object at runtime.
* Nothing but it is finally **combining** your services along with your business and generates **Businessproxy** object as a result.
* The **proxy||Weaver** class name is: **ProxyFactoryBean**.
* Factory bean classes generally will create other bean classes instances.
* Whenever you create **FactoryBean** object in Spring, instead of returning FactoryBean class object, it will be used to return that class implementation getObject() return values.
* So, this **ProxyFactoryBean** always it will return **generated runtime** **Businessproxy** bean class object. See image below:



* So, they used to call Business bean classes as: **Targets**.
* **Target**: Means nothing but your business class that is being combined with the service using **AOP**.
* In the above image, your Bank bean class is what we are referring to as; **Target**.

**NOTE:**

* From the image above; to successfully implement AOP concept, we need several packages:

1. Business bean class(s) package. 🡪 Holds your business classes.
2. Service||Aspect bean class(s) package. 🡪 Holds your Service lookup(Implementation) classes.
3. Pointcut bean class(s) package. 🡪 Holds Pointcut classes.
4. Client||Test class package.

**Contd**:

* For deposit and withdraw methods: Execute **Tx** and **Sec** services.
* But when do these 2 services need to execute?
* If it is a deposit method; **after** doing deposit or **before** doing deposit? Like what is the exact place this Tx or Sec service code need to execute?
* Okay, along with my deposit method; Tx service will execute; So, will it execute before deposit or after deposit?

**Ans:** We need to execute Tx service for deposit method **after** completion of deposit.

* Coming to security service, when does it need to execute?

**Ans:** We need to execute Sec service **before** deposit.

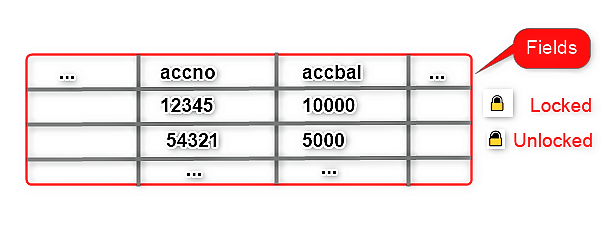
* This means that, whatever the aspects||services you lookup(Implement) in services package; these aspects||services may need to execute with your business code either:

1. **Before** a business method execution.
2. **Along** a business method execution.
3. **After** a business method execution.
4. **Before/After** business method execution.

* Hence, it will depend on the use case.
* On required conditions||Pointcuts, you need to execute these Tx and Sec services:
* If you need to execute a service||aspect **before** your business method execution:
* You need to implement that Service||Aspect lookup(Implementation) bean class from: **MethodBeforeAdvice**.
* This means that; the execution order will be:
* Service bean class logic. I.e., Security service
* Business method logic. I.e., Deposit business method.
* If you need to execute **Tx service** **after** your **deposit method** execution:
* You need to implement that Tx Service||Aspect lookup(Implementation) bean class from: **AfterReturningAdvice**.
* This means that; the execution order will be:
* Service bean class logic. I.e., **Security service**
* Business method logic. I.e., **Deposit business method.**
* Service bean class logic. I.e., **Tx service**.
* If you need to execute a service||aspect **before and After** business method execution:

I.e.,

* **Before** doing deposit, you need to send **OTP** to Mail.
* **After** doing deposit, to the Mail if you need to send Success||Failure message
* You need to implement that Service||Aspect lookup(Implementation) bean class from: **MethodInterceptor**.
* If you need to execute **Jms service**:
* When usually is it always executed?
* Whenever we get **Exceptions** from our business class methods.
* Let’s say, while doing deposit business operation: If the DB record is **locked** by another user.



* Whenever you try to perform **deposit operations** in your Bank business class into the above table to account number: 12345; then you may get some exceptions.
* Whenever you get exceptions; what do you need to do? Like, how do we need to send a message to the user? Do we need to send the raw exception||error message to the user or do we need to process that first?
* Actually, we need to process it.
* So, instead of sending exceptions||error messages to the user; Immediately that exception arises: I can keep data into **Jms lookup(Implementation) service bean class.**
* Later, I can process that Jms data into table when the lock is released.
* To achieve this requirement:
* You need to implement that Jms Service||Aspect lookup(Implementation) bean class from: **ThrowsAdvice**.
* T­­­­­­­hrowsAdvice implementations bean classes will execute in case of exceptions. Nothing but whenever an exception arises in your business methods; your Service lookup(Implementation) bean class that implements ThrowsAdvice will get executed.

Like this, there are **4 advices** to implement your Services:

1. **MethodBeforeAdvice.**
2. **AfterReturningAdvice.**
3. **AroundAdvice||MethodInterceptor.**
4. **ThrowsAdvice.**

To implement AOP concept, we have 3 approaches:

1. **Programmatic approach.**

* To go through this approach, we have discussed in details on the **4 advices** required to implement your services.

1. **Declarative||Xml approach**
2. **Annotations approach.**

**Simple AOP hands-on Ex: AOP Programmatic approach.**

In case of Programmatic AOP, we are going to learn how to:

* Implement advices in Service||Aspect bean classes
* Use Advisors.
* Prepare proxies.

**Logic**

* One simple **Hello bean class**. Inside beans package.
* Consider it as my business class.
* In AOP, we refer to business classes as **Targets**.
* For this Target class, 4 Services I need to lookup||Implement||prepare.
* You may also consider **Bank bean class**. This has the members:

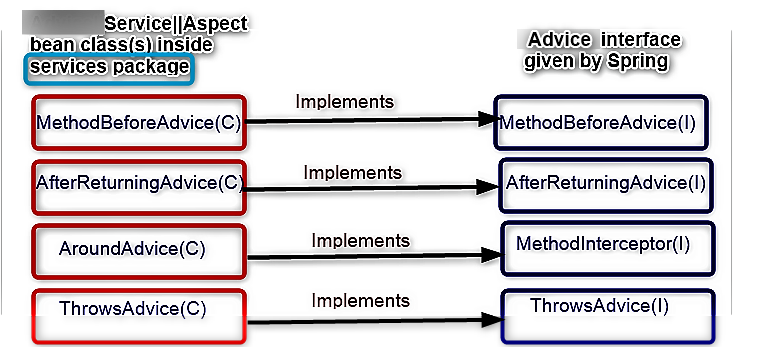
**Class Instance level variables:**

* **private int amt=5000; //** Initial **amt**
* **private String accNo=”CBA123”;**
* **deposit(int amt, String accNo)**
* It’s return type is: **int**
* Will receive 2 inputs from user.
* Check if the accNo is valid or not.
* If valid then deposit; Nothing but, **this.amt=this.amt+amt;**
* Finally, **return this.amt;**
* Else, **return 0;** if accNo is not valid.



* **Prepare Services POJO classes**; using which, we are going to lookup(implement) all the advices we’ve learnt.

See image below;



* These **Services**||Aspects that we are preparing in **services package**, need to execute for Hello bean class-**Target class**. Or any other Target class that we have prepared and need these services as per our requirement.
* Finally, write a **Test**||Client class.

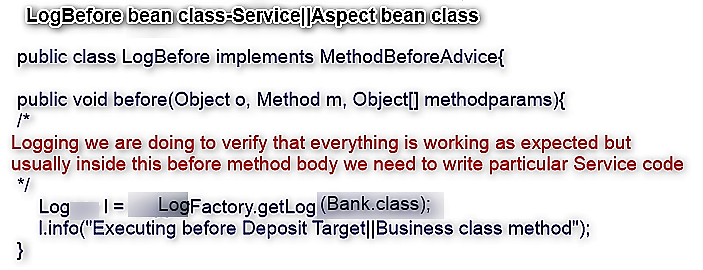
**Logic for each Service||Aspect bean class**

1. **­­­­­­­­­­­­­­­­­­­­­­MethodBeforeAdvice(C)||LogBefore(C)** 🡪 Any class name you can take as per your requirement.

* Implements from **MethodBeforeAdvice(I)**
* In this interface; there is a method: **before(-,-,-) method** that we need to override.
* This method has **void** return type.
* Takes **3** inputs||arguments:
* **Object o** 🡪 Target||Businessproxy class.
* **Method m** 🡪 **Target class** method.
* Object array of **Target class** Method input parameters; **Object[] parameters**
* Inside this method; we are going to write the **service code** we require to execute **before** a given specified business class(**Target class**) method executes.
* For this example; am just going to maintain a debugging print statement to indicate that everything will work as expected. OR, you can do some logging instead: Like some **logging service** I want to provide.
* For doing logging, just get **Log** object. i.e.,

**Log l = LogFactory.getLogger(Bank.class);** **// Same class instance you need to pass as input.**

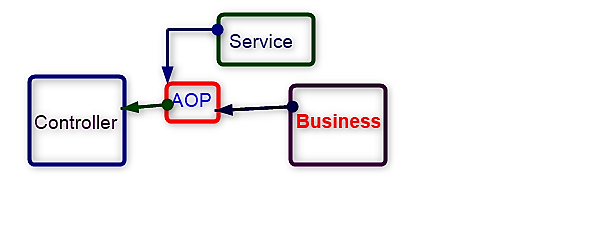
* Then, invoke the method: **l.info(“Executing Before deposit method”);** **// some info you need to pass as an argument to info() method.**
* This method, will print the **output: info + time i.e., //info 29-09-2023 Executing Before deposit method**

****

**For security** we have to use before services.

**NOTE:**

* We have our Service-**LogBefore(C)**- and Target||Business-**Bank(C)**- classes ready.
* For **Bank(C) ) deposit() method** business code will execute and:
* Will in some case return your amount and sometime it may return 0 during the course of execution.
* **LogBefore(C)** service, need to execute before your **Bank(C****) deposit() method** business code executes.
* As per architecture, who needs to call that business class: **Controller**
* But before calling this business, Controller need to get business instance along with service.
* And as we know Service and Business we can combine through AOP. So, through AOP, your Controller need to call your business.



* So AOP proxy||weaver-**ProxyFactoryBean**- will perform some weaving, and at runtime Businessproxy class object will be returned to us and that you can be able to store into Business bean class reference and access that inside your Controller class.
* To the **weaver**||**Proxy**-ProxyFactoryBean- Bank bean class is the **Target**.
* To this **Target**, you need to add a **Service**.
* How to add Service and Business to proxy||weaver? See logic and image below:

**Steps:**

1. Prepare||Create Business||Target class object. i.e.,

**Bank** **b** = **new Bank();**

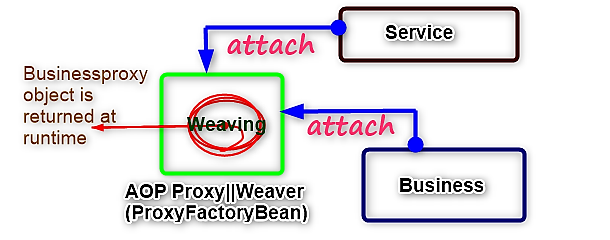
1. Create Interceptor||Service||Aspect bean class(s) object(s). i.e.,

**LogBefore lb = new LogBefore();**

1. Create AOP **ProxyFactoryBean** object

**ProxyFactoryBean pfb = new ProxyFactoryBean();**

* In this **ProxyFactoryBean** class, they have given Setter methods. Some of the setter methods include:
* **setTarget()**
* Will take the business class object reference as input.
* **setInterceptorNames(String[] names)**
* Will take String[] array of **service reference names** as input. Nothing but, String[] array of interceptor references. Interceptors are nothing but Service bean classes.



* As per the image above, we need to attach Business and Service bean classes to **ProxyFactoryBean**.
* For one business bean class, sometimes you may need to add **multiple** service bean classes.
* So, to attach a business bean class to ProxyFactoryBean; they have given: **setTarget()** setter method. This will take the business class object reference as input. i.e.,

**pfb.setTarget(b);**

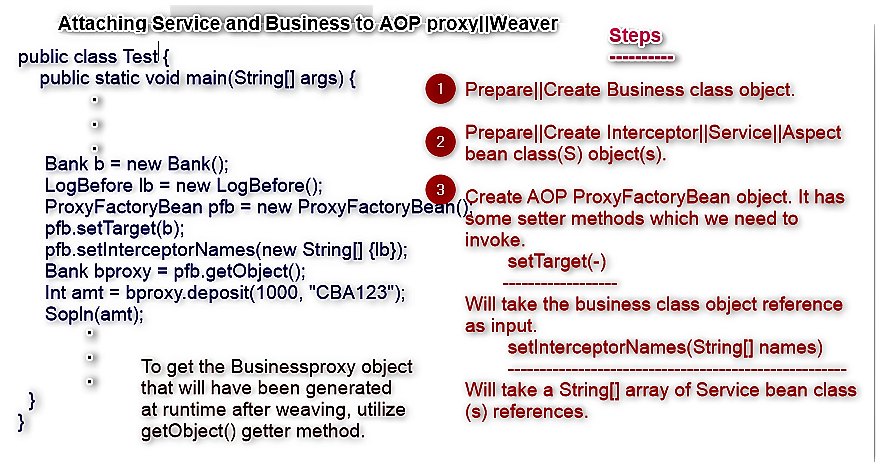
* To attach the service bean class(s)-could be multiple-to ProxyFactoryBean; they have given **setInterceptorNames(String[] names)** setter method. i.e.,

**pfb.setInterceptorNames(new String[]{lb});**

* Like this, we have added one business bean class and one service bean class to our AOP proxy-**ProxyFactoryBean**.
* From ProxyFactoryBean, we need to finally get a **Businessproxy object** that will usually be generated at runtime after weaving.
* **How to get Businessproxy object?**
* They have given one **getObject()** method
* getObject() will return us Businessproxy object.
* The generated runtime Businessproxy class, is actually child class to actual business class.
* As we know, we can Typecast child class object reference into parent class references.
* So, the returned Businessproxy object reference we can store into Business bean class reference. I.e.,

**Bank bproxy = pfb.getObject();**

* See image below for clarity:



* Now, using **bproxy**; you can be able to invoke deposit(-,-) method. i.e.,

**Int amt = bproxy.deposit(1000,”CBA123”);**

* On invoking deposit() method, the execution order will be:
* **LogBefore Service will be executed**
* **Bank class-Target class- deposit() method will now get executed.**
* In as much as we are invoking deposit() method; along with your deposit method we can be able to notice LogBefore Service class is being executed.
* LogBefore Service class Executed first then deposit() method executed next; Why? Because, LogBefore service bean class has been implemented from **MethodBeforeAdvice**. Hence, before your deposit method, LogBefore Service class will get executed.
* Incase if you implemented the same service from **AfterReturningAdvice**; then, first it will execute your business method then next it will execute your service.
* If you implemented the same service from **MethodInterceptor**; then the execution order will be:
* **Service will execute**
* Business method(s) will execute on invoking **proceed()** method.
* Service method execution will resume after proceed method has returned Target method returntype.
* If you go through, **ThrowsAdvice**; if there is any exception that arises from business method code execution, then for that exception your service will be executed.

1. **AfterReturningAdvice(C)**

* ­­­­­­­­­­­­Implements from **AfterReturningAdvice(I)**
* In this interface; there is a method: **afterReturning(-,-,-,-) method**.
* This method has a return type: **void**.
* Takes **4** inputs||arguments:
* **Object o** 🡪Takes Targetproxy class.
* **Method m** 🡪 Takes Target class method name.
* Object array of Target class method input parameters; **Object[] parameters**
* **Object returnval** 🡪 This last one takes the returned value from the target class method
* Inside this method; we are going to write the **service code** we require to execute **after** a given specified business class(Target class) method executes.
* For this example; am just going to maintain a debugging print statement to indicate that everything will work as expected. Or am instead going to perform some **Logging service**.
* So, **after** a given specified business class(Target class) method executes; if that business class(Target class) method has any return value, that return value also you can expect to be stored||inputted||injected in the 4th parameter of : **afterReturning(-,-,-,-) method**. That return value you can use as you wish.

1. **AroundAdvice(C)**

* Useful if you want to execute a Service; **before** and **after** a business||Target class method execution.

**Use case:**

* If you need to execute a service||aspect **before and After** business method execution:

I.e.,

* **Before** doing deposit, you need to **send OTP to Mail**.
* **After** doing deposit, to the Mail if you need to **send Success||Failure message**
* You need to implement that Service||Aspect lookup(Implementation) bean class from: **MethodInterceptor(I)**.
* Transactions.
* Loggings.
* etc.
* Implements from **MethodInterceptor(I)**. 🡪 one of aopalliance
* In this interface; there is a method: **invoke(-) method**.
* This method has a return type: **Object**.
* Takes **1** inputs||argument:
* **MethodInvokation** **mi**
* If we need:

1. **Object o :** Targetproxy class
2. **Object[] of target method input params &&**
3. **Method m :** Target method

for performing inspection, how can we get them? Below is how;

1. **Object o = mi.getThis();** // Will return your Targetproxy class name.
2. **Object param[] = mi.getArguments();** // It will return your target method inputs-that have been given||inputted||injected by the user.
3. **Method m = mi.getMethod();** // Will return the Target class method name.

Only for inspection the above will help us.

* Inside this method; we are going to write the **service code** we require to execute **before** a given specified business class(Target class) method executes.
* For the code that need to execute **after** a given specified business class(Target class) method executes we need to follow the following criteria:
* Call **mi.proceed()** method. On invoking this method:
* Execution will proceed to your specified business class(Target class) method code and execute that.
* Then it will return us the **return value of the target method** which we are going to store into **Object super class**.
* Immediately after this line of code, we can now go ahead and write code that need to execute **after** the given specified business class(Target class) method has executed.
* As the last line of code inside **invoke(-) method** you need to **return the return value which we just upcasted into Object super class above**.
* In **Filters**, there is f**ilter chain mechanism**; **FilterChain.doFilter()** if you call||invoke:
* **beforeDoFilter()** method code will execute **before Servlet code execution**.
* **afterDoFilter()** method code will execute **after Servlet code execution**.
* In the same way **before** your specified business||target class method execution; in the service||aspect bean class that implements **MethodInterceptor(I)**, we can maintain code that will execute at that time||point.
* When we invoke **mi.proceed()** method in the service||aspect bean class that implements **MethodInterceptor(I)**, Execution will proceed to your specified business class(Target class) method code and execute that.
* Then it will return us the **Target class method return value** which we are going to store into Object super class.
* Immediately after this line of code, we can now go ahead and write code that need to execute **after** the given specified business class(Target class) method has executed.
* As the last line of code inside **invoke(-) method** you need to **return the return value which we just upcasted into Object super class above**.
* For this example; am just going to maintain debugging print statements to indicate that everything will work as expected || am going two just implement a simple Log service.

1. **ThrowsAdvice(C) Service||Aspect class**

* Incase if any Exception arises inside your Target||Business class method, to provide **service** for these scenarios: we can do that through **ThrowsAdvice(C) Service||Aspect class**
* Like this using **ThrowsAdvice(C) Service||Aspect class,** exception services we can implement.
* ­­­­­­­­­­­­Implements from **ThrowsAdvice(I)**
* Actually, we don’t have any methods inside this **ThrowsAdvice(I)** interface at **compilation** time to be overridden, but at **runtime** AOP will add **4** implementation methods.
* This interface- **ThrowsAdvice(I),** is a simple **empty interface||Marker interface**.
* The method names are:
* **afterThrowing(-)**
* Return type is void.
* Takes an input; **Exception e**

Actually, the remaining **other 3 methods** are overloading the above method.

In the above method, the input is; **Exception** **e**

In other overloaded methods, you will find inputs like;

* **RemoteException re**
* **ServletException se**
* **…etc.**
* These methods will throw; **throws Throwable.** This in simple terms means that this method is **one for handling exceptions**.
* If this method gets any exception, here it will **throws Throwable**; **HENCE**: you will no need to handle exceptions that may arise in this method.
* So, as logic to this method; I am going to provide one print debugging statement to show that everything is working as expected. But in **real code bases;** you need to write **exception handling code** inside this method body.

**Use case of exception services:**

* **Jms cases.** Like if you want to store data into queues; and later forward that to database. This is applicable in cases of database failures or for instance in case of Saturday and Sundays; banks used to keep data into queues and then later they will process.

Like this, we have prepared our Aspect||Service lookup(Implementation)classes.

1. Assuming user passed invalid **accNo** to Bank business||target class deposit method; what happens?

**Ans**. One exception it will throw. Like **AccountNumberNotFoundException**

In case of Exceptions; if you want to execute any service-Nothing but if you want to track that user information, i.e.,

* At what time did that exception happen?

Nothing but **a log file if you want to generate**; Okay, actually Exceptions will be printed in server console. So, in case if you want to know;

* Which user given wrong accNo?
* At what time did that exception arise?

If you want to know all these details in case of exceptions. Okay from Controller if user sends any invalid accNo, your deposit method will throws one **AccountNumberNotFoundException**.

In this case if you want to track user information, nothing but if you want to do loggings incase of exceptions; you can write **service** for this scenario||use case by looking up your Service class from **ThrowsAdvice(I) interface**.

**ThrowsAdvice(I) interface**

* It is a **marker** **interface**.
* Maker interfaces means nothing but interfaces that may contain methods but implementations **may** or **maybe** **not** mandatory. But they provide **special capabilities** to your class.
* **Cloneable(I)** interface is one more example of a marker interface because it **does not force** us to implement clone() method.
* **Runnable(I)** interface is another example of a marker interface but **must** we need to override one run() method.
* Like this marker interface **sometimes it may force** us to implement some of its methods i.e., **Runnable(I)** forces us to implement **run()** method at **compile time**; but sometimes it may not force us to implement its methods at **compile time**. i.e., **Spring’s ThrowsAdvice(I)** interface and also **Cloneable(I) interface**.
* Marker interface **provides some special capabilities** to our methods.
* In **ThrowsAdvice(I)** we are **not forced** to implement any methods at **compile time**, this marker interface will provide some **special capability to your class such that it will make your class behavior execute for Exceptions**; but which method of ThrowsAdvice(I) interface will be executed at runtime? This we need the suitable method we need executed at runtime. But it is not forcing us remember.
* There are multiple methods ThrowsAdvice(I) will add at runtime to your class with dummy implementations incase you didn’t implement the methods of this marker interface; which take the following syntax:

**void afterThrowing([Method, args, target], ThrowableSubclass);**

* Whatever the ThrowsAdvice(I) method implementation you provide **explicitly** of the below listed methods; at runtime it’s **dummy** implementation won’t be provided by AOP proxy. It will leave out that method. But for the rest other methods; dummy implementations AOP proxy will provide.
* Some examples of valid methods would be:

1. **public void afterThrowing(Exception ex)**

* Only your exception it will send to us

1. **public void afterThrowing(RemoteException)**

* In case if you are making **C-S** application and if Service class is there in remote server; in that case there is a chance of getting remote exception. This means that the exception has to be remote exception.
* In that case you can go through this method.

1. **public void afterThrowing(Method method, Object[] args, Object target, Exception ex)**

* Same as first one, only that we have to use this in cases we need to perform introspection.

1. **public void afterThrowing(Method method, Object[] args, Object target, ServletException ex).**

* Go through this in case of Servlet applications and if you are getting Servlet exceptions.

**NOTE:** The first three arguments are optional, and only useful if we want further information about the **joinpoint**, as in AspectJ **after-throwing** advice.

Like this, in detail we have discussed the fundamental||Skelton knowledge about how we can **lookup**(Implement) services in Spring. We have seen how we can lookup:

* Before services.
* After services.
* Around services (Before and after)
* Exception services.

This knowledge we can use to implement **actual** services for our application.

Also, we have a Target||Business class on which we want to apply the above Services.

* Next, we need to combine the Services along with business. Using what you need to do?
* Using AOP proxies, you need to combine your services along with your Target. How to write that code?

**Logic**

* Prepare one Test||Client class.
* Inside this Client class; we need to write code that will add all the service prepared to business||Target.

**Steps**

* Create Target||Business bean class object.
* Create Services bean class objects.
* Create AOP proxy bean class object that will combine Service + Target.
* The proxy class name is: ProxyFactoryBean
* Let’s say the reference for this bean class is: **pfb**
* On top of this created AOP proxy bean class object reference, invoke a method; **setTarget().**
* This method takes Target bean class reference.
* Next, on top of this created AOP proxy bean class reference, invoke a method;

**addAdvice().**

* This method will take the Service||Aspect||Service reference that you need to add to the target.
* You can invoke this method several times to add your required services||Aspects.
* After adding Target bean class and Services to your proxy; proxy will do what internally?
* It will combine your services code along with your business code.
* Finally, invoke **getObject()** method on top of ProxyFactoryBean reference. What will happen?
* Target||Business class proxy object will be created at runtime.
* The reference to this generated runtime business proxy class will be returned.
* That returned reference we can store into parent-Business||Target bean class, reference.
* Whenever you invoke Business class methods on top of this Business bean class proxy reference; internally what output will we get and in what order?

**Output order**

1. All before Services||Aspects will be executed.

* MethodBeforeAdvice code it will execute.
* AroundAdvice **before code** it will execute.

1. Execute your Business||Target bean class method code.

* In your business||target bean class method, if any exception arises; in that case:
* ThrowsAdvice will be executed 🡪 Exception handling code will be executed. **HENCE**: The remaining **after services** below won’t be executed.

1. After Services||Aspects will be executed.

* AroundAdvice **after code** will be executed.
* AfterReturningAdvice code will be executed.

**Jar files required to test this code are:**

* In addition to whatever jar files, we have been using: **AOP jar file** is required.
* So, we require:

1. **IOC jars**

* Spring-beans.jar
* Spring-core.jar
* Spring-context.jar
* commonslogin.jar

1. **AOP jars**

* Spring-Aop.jar
* aspects.jar 🡪 For annotation-based aspects.

Actually, all we have done above; we have done it Programmatically, nothing but without using any Spring configurations and without using any Spring container.

* Using Spring Container, you should do actually. Only for understanding sake we went through simple programmatic approach without any XML configurations and also without any Spring Container.

**Hand-on XML Approach.**

**Step 1:** Create one **applicationContext.xml** file

**Step2:** All configurations we are going to perform here and Spring container will be able to understand.

**Configurations:**

* We know how to configure bean classes in Spring xml file for Spring Container to be able to create objects for them. In this case **business||Target** class ,**Service** class and **ProxyFactoryBean** am going to configure.
* Next, we have to know how to add **business||target** and **service** classes to AOP proxy object; **ProxyFactoryBean**.

**Steps:**

1. **Invoke** setter method; **setTarget** of **ProxyFactoryBean** class. How?

* Using **property** tag or **p-namespace** you can do this.
* Like this you can be able to inject business||target class to AOP proxy.

1. **Invoke** setter method; **setInterceptorNames** of **ProxyFactoryBean** class. How?

* This method accepts **input** as **String[] array of interceptor||Service** **reference names**.
* Using **property** tag you can do this. You actually know how to pass array values in Spring xml configurations. In between list and list tags, you can use value tag to pass array values.
* Like this you can be able to inject service class(s) to AOP proxy

1. Load your Spring xml file into container. Use **ConfigurableApplicationContext** container.
2. Access the weaved proxy bean class using getBean() method.
3. At the end remember to close the Container. i.e., cap.close()

In my Bank Target class; so far I have maintained only one single **deposit** method.

For deposit method I may require all the various **services** I have just configured in **app41** in this series

* Now, in case if I have **findBal()** method inside Target Bank class; for this findBal() method, I **DON’T** want any service to execute for it. Just I want to return balance-**amt**.
* For the logic of this method, I need to check condition🡪 user inputted **accNo** **accNo.equal(this,accNo)**, nothing but existing **this.accNo** then if condition has been passed successfully; return balance-**amt**. If not, throw AccountNumberNotFoundException.
* So, if I don’t want to execute any **service** for findBal() method of the target class; Bank bean class, then I need to apply conditions.
* This condition we need to implement using **point cut** classes(s).
* To implement pointcut, we have two classes:

1. **S**tatic**M**ethod**M**atcher**P**ointcut.
2. **N**ame**M**atch**M**ethod**P**ointcut.

**Note:** To be knowing more about a predefine class; be googling it.

By using these two classes, just we can apply **Pointcut**||**Conditions**. Like, where actually your service needs to execute for your business||target bean class methods.

For our Bank class bean class; two methods we have:

* deposit(int amt, String accNo): int
* findBal(): int
* Only for deposit method I want to execute my **services||advices**.
* Now am going to implement a condition class to take care of this requirement.

**Logic:**

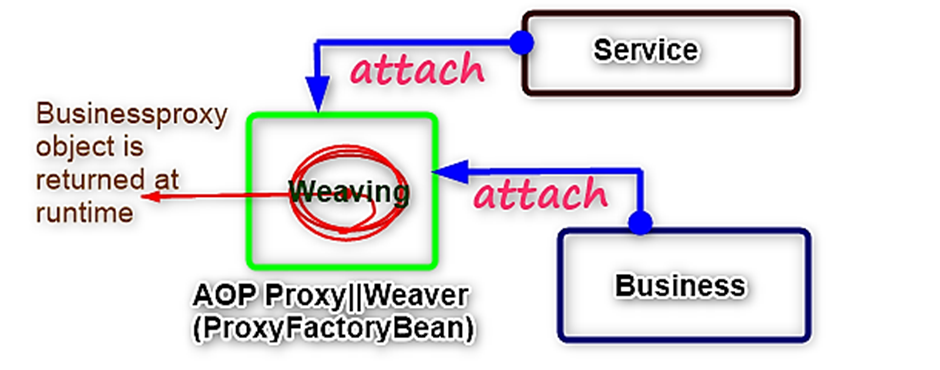
* class name: **DepositPointcut** extends **StaticMethodMatcherPointCut**.

**S**tatic**M**ethod**M**atcher**P**ointcut.

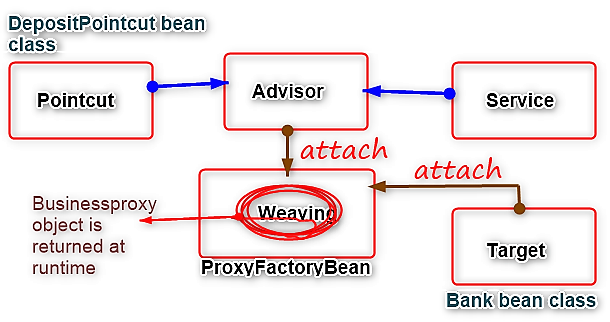
* Has a method; **matches(Method m, Class c)**
* The two inputs for **matches(-,-)** method above are for doing introspection only. Okay, here you need to check condition(s), right? Yess!!
* What condition(s)?
* matches(-,-) method will check your **target** class method names.
* If method name is fine: **return true**, else: **return false**.
* **String methodName = m.getName**();
* **If**(**methodName**.**equals(“deposit”)** : return true 🡪 Nothing but proceed to execute service.
* **else:** return false 🡪 Nothing but service won’t be executed.
* Like this our Pointcut class is ready.

**How to add Pointcut bean classes to AOP proxy: ProxyFactoryBean.**

* So far, we know how to add target and service bean classes to AOP weaver||proxy.



* To add Target bean class; we have: **setTarget() method**.
* To add Service(s) ; we have: **setInterceptorNames(String[] service\_references)**
* To add pointcut; what we should do here is:
* **Service** and **Pointcut** bean classes we have to add to **advisor**.
* We already learnt that: **Service + Pointcut = Advisor**.
* Then finally, **Advisor** you need to add to AOP proxy.
* So, our visual image would change to:



**How many Advisors do we have?**

* **DefaultPointCutAdvisor**.
* **RegexpMethodPointcutAdvisor**

For our example, we are going to go through the simple; **DefaultPointCutAdvisor**.

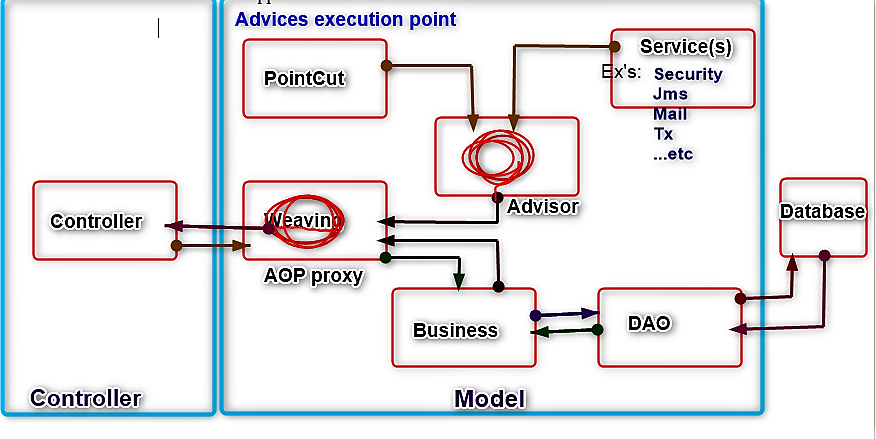
**How to add Service + Pointcut classes to Advisor.**

1. Create **DefaultPointCutAdvisor** object.

* In this class we have several setter methods:
* **setAdvice()** 🡪 Used to add Service bean class to Advisor.
* **setPointCut()** 🡪 Used to add Pointcut bean class to Advisor.

**NOTE:**

* But before calling this business, Controller need to get business instance along with service.
* And as we know Service and Business we can combine through AOP. So, through AOP, your Controller need to call your business.



**Use cases:**

* **For security** we have to use before services. (**MethodBeforeAdvice**)
* If you need to execute a service||aspect **before and After** (**AroundAdvice**)business method execution:

I.e.,

* **Before** doing deposit, you need to **send OTP to Mail**.
* **After** doing deposit, to the Mail if you need to **send Success||Failure message**
* You need to implement that Service||Aspect lookup(Implementation) bean class from: **MethodInterceptor(I)**.
* We can also use for Transactions.
* We can use for Loggings also.
* etc.
* Exception Services (**ThrowsAdvice**) you may need to use for **Jms cases**. Like if you want to store data into queues; and later forward that to database. This is applicable in cases of database failures or for instance in case of Saturday and Sundays; banks used to keep data into queues and then later they will process.