Spring Framework

Introduction:

Spring is a Framework, which is used to develop end to end applications which is par with JEE.

In a typical web application there are 3 types of logics we write, which is also referred as 3 tiers/layers.

1. Presentation-tier logic = The code/logic we write in interacting with the end-user of the application.

2. Business-tier logic = The code/logic we write in computing the output by using the data

3. Persistence-tier logic = the logic we write for storing/accessing the data from the persistence storage devices like database or file.

While working with Struts Framework, it helps us in developing Presentation-tier aspects of developing a web application, it doesn't support building business/persistence-tier of a web application.

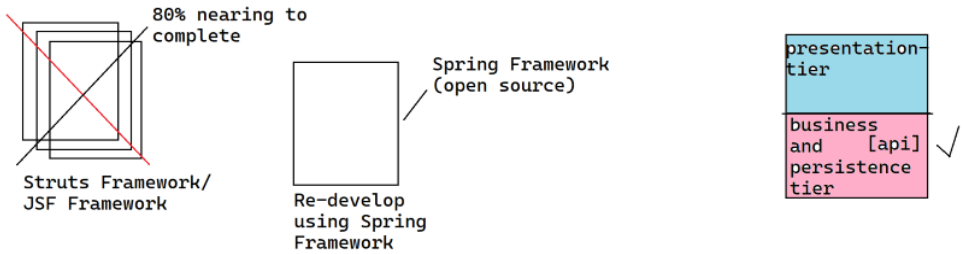
Struts is not a Framework that support building end-to-end application development, it only helps us in building Presentation-tier aspects only, so it is not a complete framework.

Unlike Struts, Spring Framework supports not only developing web application, it supports multiple application development types.

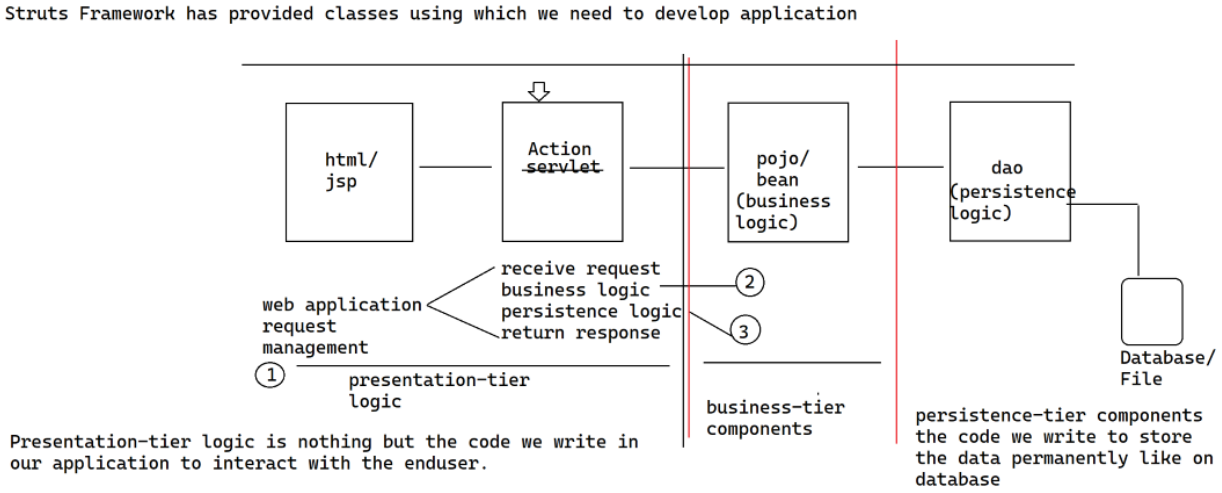
Difference between Spring and Struts?

Using Struts, we can build only web applications, and we can build only presentation-tier aspects of build a web application we cannot build end-end application using the Struts Framework.

Unlike Struts, Spring Framework supports multiple types of application development types like Standalone applications, distributed web applications, enterprise applications and Integration-Tier solutions as well. Using Spring Framework, we can build almost all the application development types, and more over it supports building end-end application,



we cannot scrapout our existing investements and re-write using Spring Framework because of Spring Framework being provided freely.



How many types of applications we can build using Spring Framework?

Whatever the type of applications we can build with JEE API, all those application development types are supported by the Spring Framework. In such case why do we need to learn Spring Framework, when we can do the same thing using JEE API.

JEE (Java Enterprise Edition) is a Standard API. API stands for application programming interface. Java Standard API's provides interfaces/abstract classes only, there will not be concreate classes. So we cannot directly work with API, we need to use implementation for that API to work with, this seems to be a very complex for a novice or a beginner to understand.

By nature, API will be huge in nature, they provide lot of classes as a part of them, since we have more number of classes, learning the API takes lot of time. These classes provided by the API are inter-linked with each other, to use a class we need to know the information about its depend, thus making the API more complex to learn and use it.

From the above we can understand the API are very complex to understand and takes more time in learning them because of huge in nature and their inter-dependencies. We cannot learn an API partially and we cannot jump start in building an application, because of inter-dependencies between the classes.

by using the Spring Framework, we can develop end-end applications, so we can consider Spring Framework as a complete application development Framework.

How many types of applications we can develop using Spring Framework?

Spring Framework support not only web application development, it supports multiple application development types like

1. Standalone Application. 2. Distributed Web Application.
3. Enterprise Application. 4. Integration-Tier etc.

Using Spring Framework, we can develop multiple application development types in par with JEE.

Then why do we need to use Spring Framework when we can develop the same type of applications using JEE?

Problems in working with API:

#1. JEE Stands for Java Enterprise Edition, it is a java standard API. API's are always partial, they provide only interfaces and abstract classes there will not be any implementation classes within them. Now to work with API we always need an implementation provided by the vendor. So by looking at API/implementations a beginner/novice always find complex to work with API.

#2. API are huge in nature, they provide lot of classes as part of them, so learning API will take more amount of time.

#3. The classes within the API are inter-dependent on each other, for e.g. to use Connection class in JDBC API we need know DriverManager similarly to use Statement class we need to use Connection for creating statement. Since the classes are inter-dependent on each other learning API and understand is very complex which takes more amount of time to learn as well, than usual.

#4. We cannot partial learn an API to develop an application, since the classes are interdependent on each other unless we understand the complete API we cannot work with, so API doesn't support quick start application development

#5. API will not provide boiler-plate logic

Boiler-plate logic: it is a piece of code that has to be written redundantly/repeatedly across various different applications we are working on is called boiler-plate logic. for e.g. while we are working with JDBC API to execute select query we write same lines of code irrespective of the query we are executing as shown below.

JDBC Sample Code:

```
Class.forName("com.mysql.cj.jdbc.Driver");

Connection con = DriverManager.getConnection(url, un, pwd);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("select * from emp");

while(rs.next()) {

  // extract data into object and use it

}
```

Now developer has to repeatedly write the same piece of code in achieving the functionality, due to which we run into lot of problems.

If API are not providing boiler plate logic, we need to write more lines of code in developing an application which will put us in more problems

1. If we are writing more lines of code in building application, we need to spend more amount of time in developing the application.

2. We need more no of developers to develop the application

3. The cost of developing the application goes high

4. The efforts of building the application is very high

5. If more lines of code, chances of increasing the bugs within our application is very high.

6. Since we have lot of code to test, the amount of time it takes in certifying or testing the application will be high.

7. The complexity in understanding the application is very high.

From the above we can understand API's doesn't support rapid application development, so go for frameworks like Spring Framework

Spring Framework supports multiple types of application development types in par with JEE. Then why do we need to learn Spring Framework to build applications when it supports same type of application development types similar to JEE.

What is a Framework?

Frameworks provides a bunch of classes, all the classes provided by the framework are concreate in nature.

What does these Framework classes will contain?

The Framework provided classes will contain pre-identified functionality, which is nothing but boiler plate logic where while developing the application with Framework we can directly use the Framework classes without writing boiler-plate logic

Advantages: -

#1. Since Frameworks directly provides concreate classes to us, the complexity of API/implementation is gone, now developer can directly work with Framework classes.

#2. Unlike API Framework will have less number of classes, so learning Frameworks is quick when compared with API.

#3. Since they provide concreate classes, there is no or minimal inter-dependency between the classes, so it is obvious that those are less complicated and quick to learn and use it.

#4. Framework supports jump start in building an application which means, we can partially learn a Framework and can begin development of the application.

#5. Framework provides boiler-plate logic, thus the number of lines of code we need to write in building the application is very less when compared with API.

5.1 it takes less time and resources in building an application.

5.2 efforts of developing the application is very less when compared with API

5.3 cost of building the application comes down

5.4 chances of bugs in the application will be very less due to less number of lines of code

5.5 The framework developers have written the boiler-plate logic by adopting the best-practices and optimizations in implementation, so while using the Framework all those best practices would come to our project as well, so we are indirectly getting benefited

5.6 Since we are writing less number of lines of code, the time required for testing/certifying the application will be less.

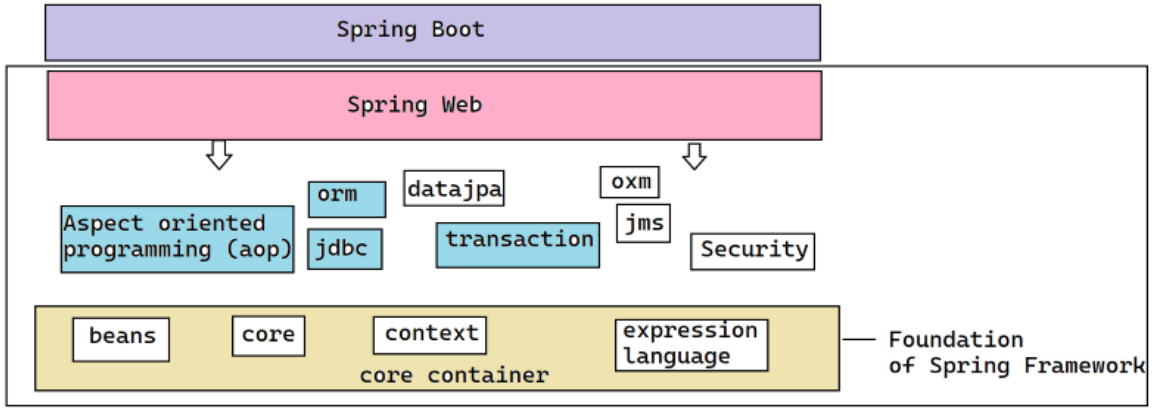
From the above we can understand frameworks support rapid application development.

Spring Framework even though it is very big, still it is being called as Light weight application development Framework, why is it being termed as light weight application development framework?

Even through Spring Framework supports lot of application development types, still it is being considered as light weight application development framework, because Spring Framework developers has not served the Spring Framework as one Single distribution. The Spring developers to make the Spring Framework learn quickly and easy to use, they broken down the Spring Framework into parts called "**Module**". While breaking the framework into modules, they ensured each module is at most independent of another module, so that the developers never need to worry about how big it is, rather they only need to think about what they want in it and how to use that module. So if we can use what we want out of the Framework without worrying about how big it was we can obviously say it is

"**Light-weight application framework** ".

Let us try to understand the architecture of the Spring Framework to judge it as light weight application development framework.



How do we support Spring Framework is a light weight application development framework?

Spring Framework has been broken down into multiple modules where each module is independent of another.

The foundation or fundamental modules on which other modules of the Spring Framework work is core container which has 4 modules inside it

1. core
2. beans
3. context
4. spring expression language.

To work with Spring Framework, the developer doesn't have to learn the entire Spring Framework and should use it, instead depends on the type/nature of the application he has identify the relevant module, can learn and use along with core container the specific module of the Spring Framework.

Thus making Spring Framework light weight, since he can choose and use the module of his choice only.

What are the challenges are encountered by the Spring Framework initially? How does Spring Framework overcome them?

1. Struts Framework by then has been used by lot of people in the market and it has a huge customer base and many of the people were aware of it, being new Spring Framework, no one knows about it and doesn't have any customer base. Due to this most of the people shows interest in building their application using Struts Framework rather than Spring Framework.

2. There are lot of applications developed on Struts Framework and were successfully deployed onto production, whereas Spring Framework is a newbie, no one has used it and there are no successful deployments made through Spring Framework. So there is no guarantee that people can develop application and can delivery using Spring Framework, so most of the people go for Struts Framework.

3. By the time Spring Framework got release, the Struts Framework has got multiple releases and most of the bugs in the Struts Framework are already fixed and turned to be a Stable Framework, so if we choose Struts Framework in developing our application there is a guarantee we can complete our application development. Spring Framework just got release, it may still have bugs if we use Spring Framework there is no guarantee we can build our application, because there might be some bugs in Spring Framework which might block our application development

4. Struts Framework can be considered as next door boy, where while working with Struts Framework if got stuck somewhere, we can always find someone around us who can help us in resolving it, whereas Spring Framework being newbie, we cannot find someone who can jump in unblocking us, which is biggest problem.

What are the challenges in front of the Spring Framework in their initial days, to overcome in order to with stand in the market?

1. Struts Framework exists a way before Spring Framework, it has huge customer base and lot of users are already using the Struts Framework, when it comes to Spring Framework it has little or no presence in the market. Looks like most of the users are going to use Struts Framework only for their application development due to its presence and popularity

2. There are lot of application that are already developed and deployed in production environment using Struts Framework, looks like Spring Framework is a newbie where there are very less or no application are built on it and deployed in production. Since there are no use cases running in production, there is no guarantee that by using Spring Framework we can deliver the application into production

3. By then Spring Framework has been released there are multiple releases of Struts Framework taken place, which made it more stable. Whereas Spring Framework being newbie there are no release of the Spring Framework happened and there is no guarantee all the bugs in the Spring Framework are eliminated, so there is a risk involved in building an application using Spring Framework, during the development if we encounter any critical bug in Spring Framework, our development will be halted.

4. We can always find someone who can help us in resolving the problem, when we stuck while working with Struts Framework, because it is being used my many people in the market. whereas Spring Framework being newbie looks like most of the people don't know, if we stuck somewhere while developing application using Spring Framework we don't get any references barely that further delays the application development.

5. There is a huge community support and rich amount of documentation is available with Struts Framework being established very well, so we can always find it easy to develop the application using Struts Framework. Spring Framework is a newbie, there is no community support or documentation available in the market due to which always it is quite difficult to develop application

6. Struts Framework and Spring Framework are open source, being open source always there is a risk involved. The open source technologies might exist and may sunset by themselves at any time without providing the support or future releases due to which all the applications being developed using the open source stack will be effected. Struts Framework exists from long time and lot of people are using the Struts Framework there is guarantee struts would continue to exists for long time. whereas Spring Framework being newbie there is no guarantee it would exist for long time.

Looks like Spring has quite a number of challenges in front of them, to be popular like Struts Framework Spring should address all the above challenges as well.

- ✓ Spring Framework has added spring community support
- ✓ provided better documentation to help
- ✓ quick release of the Spring Framework made it more active and stable
- ✓ Spring Framework developers has conducted lot of technical conferences in the market
- ✓ Spring Framework built use cases and deployed on production to give market more confidence about them and partnered with many assignments helping them to make production

#1 Struts Framework support developing Web Applications only and we can develop only the presentation-tier aspect of a typical web application, we can consider Struts as not a complete application development framework.

Unlike Struts Framework, Spring Framework supports developing multiple application development types like Standalone application, Distributed Web Application, Integration Application and Enterprise Applications.

Spring Framework supports building all the aspects/tiers of an application, so we can consider Spring Framework as an end-to-end application development framework.

#2 Spring Framework supports developing all the application development types in par with JEE.

#3 Spring Framework is a very huge/big as it supports lot of application types, but even then also it is considered as light weight application development framework.

The Spring Framework is considered as light weight application development framework due to its modularity in nature. The Spring has been broken down into several modules where each of these modules are completely independent of each other. The only module on which all the modules Spring Framework are dependent is core container (core, beans, context, expression language).

So to work with Spring Framework the developer doesn't have to learn the entire spring and no need to use it as a whole, based on the requirement he can choose an appropriate module of the Spring Framework learn it and use for developing the application, thus making it completely light weight because I don't need to bother anything apart from what I need.

What are the challenges Spring Framework having when it was introduced in market?

1. No market footprint for Spring Framework, but there is a huge customer base already exists for Struts Framework

2. There are lot of use cases successfully deployed into production by developing through Struts Framework, whereas Spring is newbie and there are no use cases being deployed in production

3. Struts Framework is more stable without bugs, whereas Spring Framework has just then released

4. More community and documentation is available for Struts Framework, whereas Spring doesn't have

5. Struts is next door boy, where we can find many people around us who knows Struts Framework, but Spring Framework being new, we don't find anyone around us to get help

6. Struts and Spring Framework are open source, but Struts has a guarantee of existence, whereas Spring being newbie there is a risk involved that it might be removed

---

There are 2 key features of Spring Framework that makes spring unique in the market.

1. versatile application development framework
2. non-invasive application development framework

There are 2 key features of the Spring Framework that made Spring unique and more successful

1. Versatile application development framework

Spring Framework is very flexible enough in integrating with any of the existing technologies with which we build the application, so that we don't need to rewrite the entire system to use Spring Framework rather with little changes in our application we can integrate Spring and enrich the entire system.

2. Non-Invasive application development framework

usually while working with an API/framework, we need to use API/framework provided class within our application classes to build the functionality/application.

For e.g. to write an Action class in struts framework

```
class EmployeeAction extends Action {

        public ActionForward execute(HttpServletRequest request, HttpServletResponse, ActionForm,
        ActionMappings) {

                // logic

        }

}
```

Here, Action, ActionForm, ActionForward, ActionMappings are the classes provided bythe Struts Framework, in our code we are using Framework/API classes to use that framework or API.

Here our code is tightly coupled with Framework/API since we are using Framework/API classes directly within our code. let us say we want to remove Struts Framework in our application and want to use a different framework, now how to separate the application from Struts Framework?

go to each class of our application and identify where we are using Struts Framework classes and remove the references of them, looks like we need to rewrite the entire code of our application since we are using Struts Framework everywhere.

This is called invasive, which means the framework or API code will be creeped into our code.

While working with Spring Framework, we don't need to write our code either by extending/implementing or referring Spring Framework classes. So our code is loosely coupled from Spring Framework, but still we can use the functionality of the Spring Framework. The real benefit is to separate our application from Spring Framework we need to modify zero lines code of our application, which is called non-invasive application development framework.

What does Spring Core module offer?

we are going to build an application, by writing several classes, these classes can be broadly classified into 3 types based on the nature of the code we are writing with them, those are as below.

1. POJO = plain old java object

if we can compile/run the code without using any of the external third-party libraries under classpath, then the class is called POJO class.

2. java bean

A class written with attributes, it should have 0-arg constructor and setters and getters for all of the attributes declared in that class, then the class is called java bean.

```
class Bike {
        int bikeNo;
        String manufacturer;
        double price;
        public int getBikeNo(){}
        public void setBikeNo(int bikeNo){ }
        public String getManufacturer() {}
        public void setManufacturer(String manufacturer) {}
        public double getPrice() {}
        public void setPrice(double price){}
}
```

3. bean (or) component

A bean or component class is a general purpose class which may contain attributes and methods, the methods contain arbitrary logic in performing operation, such type of classes are called "bean" or "component".

An application is build out of several classes, each of them performing various different types of functionality. Based on the nature of code written inside the classes, we can broadly classify the classes into 3 types.

In an application the bean or component classes plays a major role in building the application.

we can write a class with any amount of lines of code, but to avoid maintainability problems it is often recommended to break the code into multiple classes.

What type of maintainability problems we run into?

1. complexity in understanding the code
2. debugging is very difficult
3. we cannot reuse the code in other parts of the program

we can achieve reusability through modularity.

always break the code into multiple components so that we can reuse a part of the code in another place where ever we need.

From the above we can understand we need to modularize and distribute the code into multiple bean or component classes within an application.

Every class cannot be complete by itself, it may have to use or refer or talk to some other class to complete its functionality. So a class may be dependent on another class to complete its functionality.

How to manage dependency between the classes?

In one class we need to write the logic to talk to another class, so that it can complete its functionality.

Instead of we managing the dependencies between the classes, spring core takes care of managing the dependencies for us.

- Spring core is a module, that help us in managing the dependency between the classes.

```
class A {
  void m1() {
    B b = new B();
    int j = b.m2();
  }
}
class B {
  int m2() {  }
}
```

The majority of application logic will be written as part of component classes of our application those places a crucial role in building an application.

We should not write the entire application logic into one or few component classes, we need to break down and distribute the code into adequate number of component classes for readability, maintainability and reusability.

From the above we understood we broke the code into multiple component classes, so looks a component class may need to talk to another component class to complete its functionality.

A class cannot be complete by itself, it may have to talk to some other class within our application to complete its functionality, which are called dependent classes.

How to manage dependency between the classes?

The Spring Core is all about managing the dependencies of the classes.

What is Spring Core?

Spring core is a module that help us in managing the dependency between the classes. Spring core can manage dependencies between any 2 arbitrary set of classes, but it provides recommendations in designing the classes, so that those can be better managed through spring core and we take lot of advantage of using Spring Framework.

Spring Core recommends us to design our application classes based on Strategy Design Pattern, so that we can get more benefit of using Spring Framework in developing our application.

What is Spring core being about and, what does it provide?

Spring core is all about managing the dependencies between the classes.

Even though Spring core can manage dependencies between any 2 arbitrary set of classes, it recommends us to design the classes based on the Strategy Design Pattern and give to him. So, that it can better manage the dependencies and we can get benefited out of using Spring Framework.

What is a Design Pattern?

For a recurring problem, there is pre-computed best solution that can be applied under a specific context to solve the problem together documented is called design pattern.

The roots or notion of design patterns in the software engineering world has been started by 4 people Richard Helm, Ralph Johnson, Enrich Gamma, John Vissel who are popular know as Gang of Four. The document the recurring problems and their best applicable solutions under a context and published a book called "Elements of Reusable object oriented Software".

These people have document the problems and solutions that we generally face while building the application on object oriented programming principles.

Strategy Design Pattern is one of the design pattern out of the patterns of GOF, that Spring people recommends us to use in design the application classes, to get more benefited.

Strategy Design Pattern:

The strategy design pattern has provided 3 principles based on which we need to design our application classes.

1. Favor composition over inheritance
2. always design to interfaces, never design to concreate classes
3. your code should be open for extension and should be closed for modification

---

#1. Favor composition over inheritance

If a class wants to reuse the functionality of another class, there are 2 ways are there Inheritance & Composition

Inheritance: -

We can establish Inheritance relationship between the classes by extending one class from another.

eg.

```
class A extends B {

}
```

When we inherit one class from another class, all the traits of the parent will be part of the child class. The methods of the parent can be called by child class method directly as if those methods are also part of the child without using the object of parent.

Inheritance always establishes IS-A relationship between the classes. IS-A relationship means always the child can be expressed in terms of Parent.

Spring core is all about managing the dependency between the classes. Spring core can manage the dependencies between any 2 arbitrary set of classes, but it recommends us to design the classes based on a design pattern called "Strategy Design Pattern", to get most benefited out of using Spring Framework.

For a recurring problem, there is a best applicable solution that can be used/applied to solve the problem, under a context documented together is called a "Design Pattern".

Strategy design pattern is one of the design patterns defined as part of GOF Patterns.

We need to design our application classes based on Strategy Design Pattern, and pass them to Spring Framework to better manage the dependencies.

```
class B {
 int m2(int i) {
  // operation
  return 34;
 }
}
```

```
class A extends B {
 void m1(int i) {
  int j = m2(i);
  // operation
 }
}
```

when we inherit a class from another class all the traits of the parent class will be part of child. So that child can directly use the methods or attributes of the parent as if those are part of child only.

Here A class can call the method m2(int i) of B class without using the object of B class, because A is having the traits of B inside it.

Inheritance establishes IS-A relationship between the classes, which means always a child can be expressed in terms of parent.

When we go for inheritance in reusing the functionality of another class, all the traits of parent will be inherited (part of) to child class. we use extends keyword in deriving the inheritance relationship. Inheritance establishes "IS-A" relationship between the classes.

IS-A:

The child contains all the features of the parent and we can see a child as a substitution of the parent as he poses everything of his parent.

(or)

A child can be expressed in terms of the parent.

#2. What is composition?

The another way we can reuse the functionality of another class inside a class is through "Composition". We declare another class as an attribute inside our class, instantiate the object of other class and use the methods and attributes of other class through the reference of the other.

```
class A {
 B b;
 void m1(int i) {
  b = new B();
  int j = b.m2(i);
  // operation
 }
}
```

```
class B {
 int m2(int i) {
  // operation
  return 93;
 }
}
```

Composition always refer to a thing/or a substance has been made up of what other things. For eg.. a Bicycle is made up on lot of other parts like break, chain, frame, handle etc.

Composition establishes HAS-A relationship between the classes, which means we have the reference of other to use the functionality of other class.

when to use inheritance, and when we should go for composition?

Inheritance: -

1. If we want to reuse all the traits of another class, within our class then use Inheritance.

2. if we can replace a parent with the reference of child, which means always a child can act as a substitute of the parent. (nothing but child is-a parent)

```
class Printer {
 - init() {}
 - print() {}
 - diagnose() {}
}
```

```
class LaserJetPrinter extends Printer {
        // here child can be expressed interms of parent
}
```

```
class Engine {
  void ignite() {}
  void accelerate(int kmph) {}
  void break() {}
}
// this is wrong, because we cannot replace a Engine with Car. It is not a IS-A relationship.
class Car extends Engine {
}
```

When to go for composition?

If we want to partially use the functionality of another class inside our class, then use Composition.

When to go for Inheritance:

There are 2 thumb rules on which we need to choose Inheritance

1. In our class we want to reuse all the methods of another class

2. our class can be seen as a substitute of another class

then we need to go for inheritance

What is a design pattern?

For a recurring problem under a context, there is best applicable solution that can be applied to solve the problem together documented called "design pattern".

GOF design patterns document problems and the solution we encounter while building the applications on object oriented programming languages. one of the design pattern out of GOF patterns is "Strategy Design Pattern".

Strategy Design pattern is recommending us to use Composition, rather than Inheritance most of the time because there are few challenges we will encounter if we use Inheritance, to avoid them it is recommending us to use Composition.

What are the challenges or difficulties of using Inheritance?

Problems or challenges of using Inheritance:

1. In most of the real-time use cases, a class wants to use few of the functionalities of another class, but not all of them. In such case it is recommended to use Composition only than inheritance. By which we can understand most of the real-time use cases are solvable through composition rather than inheritance.

2. Most of the programming languages including java doesn't support multiple inheritance. In a case where if your class wants to reuse the functionality of multiple other classes the only way we can accomplish is through Composition, we cannot use Inheritance because of the above limitation.

3. The classes will become fragile if we go for inheritance, if we use composition those are less fragile.

```
class A {
 int m2(int i) {
  // operation
  return 23;
 }
}
```

```
class B extends A {
 int m2(int i) {
  int j = super.m2(i);
  // perform some
additional operation
  return 24;
 }
}
```

A a = new B();

always to the parent class reference variables we can assign any of the child class objects, through which we can achieve runtime polymorphism.

int k = a.m2(10);

In general java determines the method to be called on a class based on the reference type we are using in calling the method.

But whenever we assign a derived class object to a base class reference variable, java will not call the method based on the reference type, rather it checks to see the reference variable is pointing to which class object and calls the method on that corresponding class, this is called "Runtime Polymorphism". Here the method to be called on which class whether it is "A" or "B" class is determined only at runtime because objects are instantiated and assigned to variables at runtime only.

Without modifying the code inside m2() method of class A, we can replace the logic or method m2() of class A through overriding. If we see even though we are calling m2() method using reference variable A still the method of B class m2(){} method is called, which looks like we are invoking new m2() using variable A (indirectly means replacements);

```
class A {
 double m2(int i) {
  // operation
  return 23.23;
 }
}
```

```
class B extends A {
 int m2(int i) {
  int j = super.m2(i);
  // perform some
additional operation
  return 24;
 }
}
```

```
class C {
 void m3(int j) {
  B b = new B();
  int k = b.m2(j);
 }
}
```

What is overriding?

overriding happens between the classes which are under inheritance relationship. A method to participate in overriding, the name of the super class method and subclass method should be same including parameters and return Type.

What is overloading?

overloading may happen within the classes or across the classes of inheritance hierarchy. A method is said to be overloaded if the method name should be same, but parameters should be different between those 2 methods. but it doesn't consider return Type in overloading.

2#. Always design to interfaces, never design to concreate classes

If a class cannot be completed by itself and if it has to talk to another class to fulfill its functionality, then the class is said to be dependent on another class.

How to manage dependency between the classes?

There are 2 ways of managing the dependencies

1. Inheritance

2. Composition

based on the 1st principle of strategy design pattern we learnt inheritance is not recommended, so we need to go for composition only.

```
class A {            |   class B {
  B b;               |    int m2(int i) {
  void m1(int i) {   |      // arbitary logic
    b = new B();     |      return 102;
    int j = b.m2(i); |    }
  }                  |  }
}                    |
```

In the above eg we are using composition only, here class A is holding the concreate reference of class B in talking to the class B, so that a change in class B effects the class A directly and these 2 classes are said to be tightly coupled with each other.

coupling = inter-dependency between the classes of our application is called coupling

cohesion = A class complete by itself is called cohesion

Always avoid coupling between the classes and encourage cohesion.

There are lot of problems if the classes are tightly coupled with each other.

A change in one class will affect lot of other classes within our application due to which we run into several problems

1. as we are modifying several classes within our application the time required for making the changes is very high

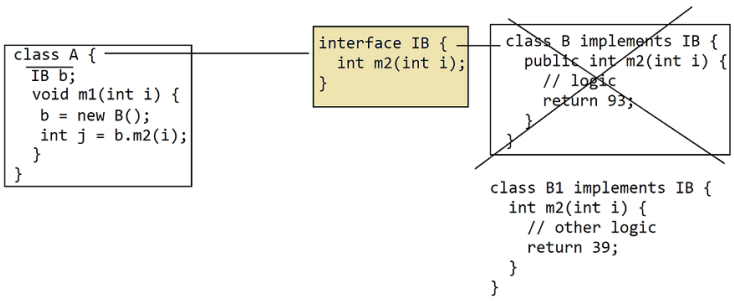
2. the cost of making the changes is going to be very high

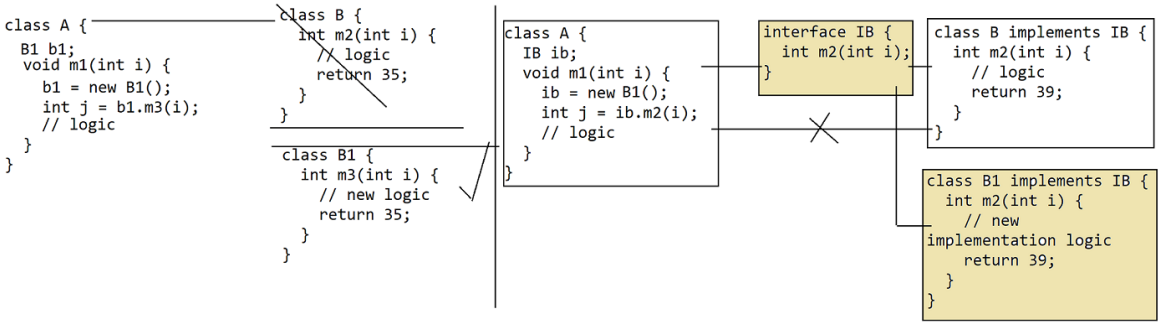
3. as we are modifying several classes within our application, there is always a high risk of introducing bugs because of the changes

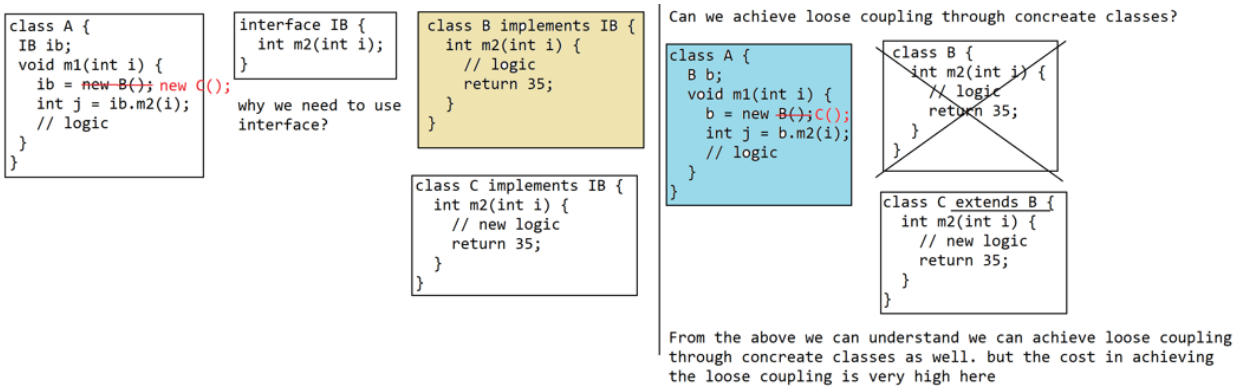
4. testability and certifying the changes is going to take lot of time

overall maintainability of the application becomes very complex as a small change impacts several parts of the program. So avoid coupling between the classes

From the above we learnt while using composition, the classes are said to be tightly coupled with each other. So how to avoid coupling between the classes?


class A {
IB b;
void m1(int i) {
b = new B();
int j = b.m2(i);
}
}
interface IB {
int m2(int i);
}
class B implements IB {
public int m2(int i) {
// logic
return 93;
}
}
class B1 implements IB {
int m2(int i) {
// other logic
return 39;
}
}


class A {
B1 b1;
void m1(int i) {
b1 = new B1();
int j = b1.m3(i);
// logic
}
}
class B {
int m2(int i) {
// logic
return 35;
}
}
class B1 {
int m3(int i) {
// new logic
return 35;
}
}
class A {
IB ib;
void m1(int i) {
ib = new B1();
int j = ib.m2(i);
// logic
}
}
interface IB {
int m2(int i);
}
class B implements IB {
int m2(int i) {
// logic
return 39;
}
}
class B1 implements IB {
int m2(int i) {
// new
implementation logic
return 39;
}
}


class A {
IB ib;
void m1(int i) {
ib = new B(); new C();
int j = ib.m2(i);
// logic
}
}
interface IB {
int m2(int i);
}
why we need to use interface?
class B implements IB {
int m2(int i) {
// logic
return 35;
}
}
class C implements IB {
int m2(int i) {
// new logic
return 35;
}
}
Can we achieve loose coupling through concreate classes?
class A {
B b;
void m1(int i) {
b = new B();C();
int j = b.m2(i);
// logic
}
}
class B {
int m2(int i) {
// logic
return 35;
}
}
class C extends B {
int m2(int i) {
// new logic
return 35;
}
}
From the above we can understand we can achieve loose coupling through concreate classes as well. but the cost in achieving the loose coupling is very high here

#2.3 Why does the new implementation we write should implement from the same interface

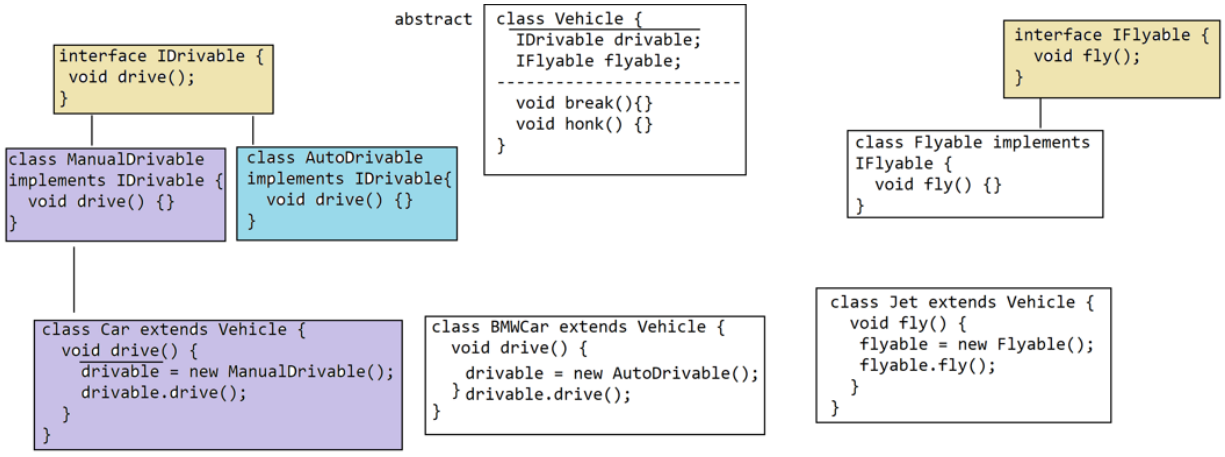
```
interface IB {      class B implements IB {
 int m2(int i);      int m2(int i) {
}                     // logic
                      return 93;
                    }
                   }
```

now we want to write new algorithm approach of building the functionality than the logic in B class. So we are coming up with C class, if it has to serve the same functionality of B class, then C also should fulfill the same interface of B

```
class C implements IB {}
```

#2.4 Why do we need to use interfaces only to achieve loose coupling why not concrete classes?

We can achieve loose coupling through concrete classes



#2.5 why do we need to go for interfaces for achieving loose coupling, why not abstract classes?

Of course we can achieve loose coupling through Abstract classes as well shown below.

```
class A {                    abstract class AbstractB {      class B extends AbstractB {
  AbstractB abstractB;           abstract int m2(int i);         int m2(int i) {
  void m1(int i) {             }                                 // logic
    abstractB = new B();                                         return 3;
    int j = abstractB.m2(i);                                   }
    // logic                                             }
  }
}
```

A can talk to any of implementations of the AbstractB, not only to the B class above

class C extends AbstractB {}, then A class can talk to C as well. So we are able to achieve loose coupling through Abstract classes.

In general we use Abstract classes if we have partial implementation to be reused across multiple classes. here the AbstractB has no partial implementation, its an pure abstract class, by which there is no point in creating as an Abstract class rather we should go for interface only. If we have any partial implementation that can be shared then we can use Abstract class as well.

```
class A {
  AbstractB abstractB;

  void m1(int i) {
    abstractB = new C();
    int j = abstractB.m2(i);
    // logic
  }
}
```

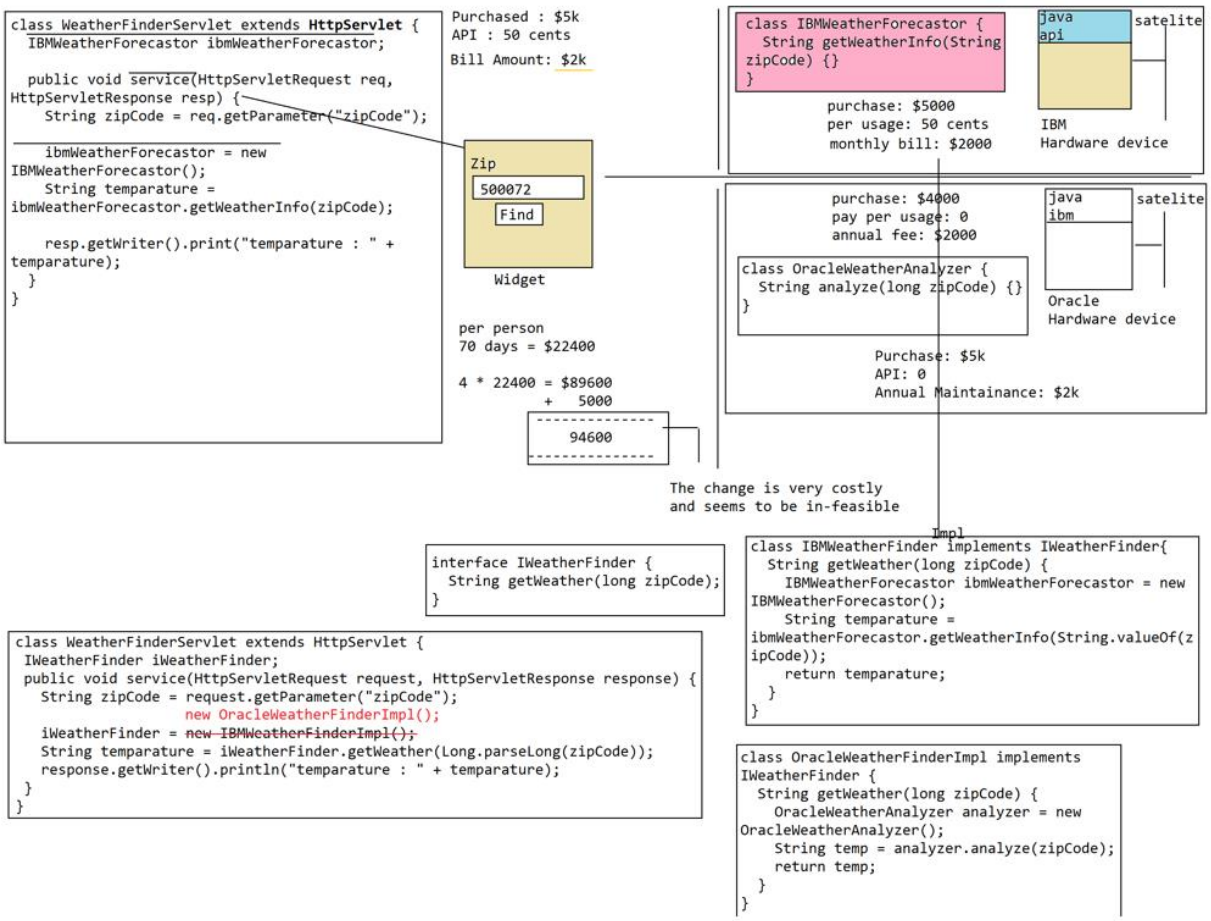
```
abstract class AbstractB {
  abstract int m2(int i);
}
```

```
class B extends AbstractB {
  int m2(int i) {
    // logic
    return 39;
  }
}
```

```
class C extends AbstractB {
  int m2(int i) {
    // logic
    return 34;
  }
}
```

#3 code should be open for extension and closed for modification

Inheritance makes the application design rigid.



By designing the classes based on the above principle our classes will become completely loosely

Example:

```
class MessageWriter {
  IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    messageFormatter = new PDFMessageFormatterImpl();
    cMessage = messageFormatter.format(message);
    System.out.println(cMessage);
  }
}
```

```
interface IMessageFormatter {
  String format(String inMessage);
}

final class HTMLMessageFormatterImpl implements IMessageFormatter {
  final public String format(String inMessage) {
    return "<html><body>"+inMessage+"</body></html>";
  }
}

final class PDFMessageFormatterImpl implements IMessageFormatter {
  final public String format(String inMessage) {
    return "<pdf>"+ inMessage +"</pdf>";
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = new MessageWriter();
    messageWriter.writeMessage("Welcome to sdp");
  }
}
```

How to create a maven project?

Open command prompt and navigate to any directory **ex:** d:\workspace\spring\core:/>

mvn archetype:generate -DgroupId=core -DartifactId=strategydp -Dversion=1.0 -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4

The project will be created with the below directory structure:

```
d:\workspace\spring\core:/>
  strategydp
  |-src
    |-main
      |-java (sourcecode)
      |-resources (sourcecode, non-compile)
        |-*.xml
        |-*.properties
    |-test
      |-java (testclasses sourcecode)
      |-resources (test sourcecode non-compile)
  |-pom.xml (project object model) = information about the project
  |-target
    |-classes
      |-*.class
    |-strategydp.jar
```

```
class MessageWriter {
  private IMessageFormatter messageFormatter;

  public void writeString(String message) {
    String cMessage = null;

    messageFormatter = new PDFMessageFormatterImpl();
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
```

```
  }
}

interface IMessageFormatter {
  String formatMessage(String message);
}

final class HTMLMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<html><body>"+message+"</body></html>";
  }
}

final class PDFMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<pdf>"+message+"</pdf>"
  }
}



class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = new MessageWriter();
    messageWriter.writeMessage("Welcome to SDP");
  }
}
```

In the above example even though MessageWriter is talking to HTMLMessageFormatterImpl through the help of interface still they are not completely loosely coupled. In order to switch from HTML to the PDF format we need to still modify the instantiation logic in creating the object of another class. So we can say our class has still some amount of coupling being left.

Even though we used interface, why there is a coupling?

1. If a class wants the object of another class, in general our class has to create the object of another class.

For eg. if class A wants the object of class B then inside the A we have to write the logic for creating the object of B.

In order to create the object of another class, we need to use new operator and has to pass concreate class name of another class in our class. if we are using concreate class name to switch from one class to another we need to change the class name again which means our classes are tightly coupled.

2. If a class is creating the object of another class means, our class will be exposed to the complexity in creating the object of another class.

If the instantiation of another class requires complex logic, then all the classes whichever want the object of another class has to write the same complex logic in instantiating the object of another class, so that the instantiation logic will be duplicated across all the classes. If there is change in instantiation logic, then all the class will be impacted.

```
class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    cMessage = messageFormatter.formatMessage(message);
    System.out.printl(cMessage);
  }
}
interface IMessageFormatter {
  String formatMessage(String message);
}

class HTMLMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<html><body>"+message+"</body></html>";
  }
}
```

```
class PDFMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<pdf>"+message+"</pdf>";
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = new MessageWriter();
    messageWriter.writeMessage("Welcome to SDP");
  }
}
```

If a class is creating the object of another class, then we will run into 2 problems.

1. In order to create the object of another class, our class has to use the new operator and has to refer the concrete class name of another class, to switch from one class to another class again we need to change the concrete class name of another which makes our class tightly coupled.

2. if a class is creating the object of another class, then it is exposed to the complexity in creating the object of another class.

```
class A {          class B {      class C {}
  void m1() {        C c;
                   B(C c) {}
  }              }
}
```

In the above eg class A want the object of B class, but in order to create the object B A has to know the details of how to create the object of B (like B is dependent on C etc) and should write complex logic in instantiating the object of B class.

C c = new C();

B b = new B(c);

If we are writing that complex logic in creating the object of another class, we will run into maintainability problems

1. the code in creating the object of another class would get duplicated across all the classes whoever want the object of B here
2. if instantiation of the B class has been changed again our class will gets affected.

If our class is creating the object of another class we are running into the above 2 problems, in order to overcome the problem, let our class don't create the object of another class. Let some other create object of the another class.

Factory design pattern

Factories are used for creating the object of another class.

There are 2 reasons for which we use the factories.

```
class MessageWriter {
private IMessageFormatter messageFormatter;

public void writeString(String message) {
  String cMessage = null;

  messageFormatter = new HTMLMessageFormatterImpl();
  cMessage = messageFormatter.formatMessage(message);
  System.out.println(cMessage);
}
```

```
}

interface IMessageFormatter {
  String formatMessage(String message);
}

final class HTMLMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<html><body>"+message+"</body></html>";
  }
}

final class PDFMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<pdf>"+message+"</pdf>"
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = new MessageWriter();
    messageWriter.writeMessage("Welcome to SDP");
  }
}
```

If a class is creating the object of another class, we will run into 2 problems

1. to create the object we need to use new operator and need to pass concreate classname, so that we will tightly coupled with the classname of another class.

2. we will be exposed to the complexity in creating the object of another class, so that if instantiation process has been changed for other class, we will be impacted.

How to overcome both the problems?

Factory design pattern

Factories are used for creating the object of another class, there are 2 reasons why we use factory class.

1. Factories helps us in abstracting the complexity in creating the object of another class.

Every class cannot be instantiated by using new operator, there are few class who requires complex instantiation process in creating the object.

for eg..

```
class A {                class B {        class C{}  class D {}
  B b;                   C c;
  void m1() {              D d;
    want object of B       B(C c, D d){}
  }                    }
}
```

here if A wants the object of B, then A has to create the object B, but in order to create the object B he has to create C class and D class objects and pass to B, where in A has to know the complete details and has to write lot of lines of code in creating the Object B, which is complex instantiation process.

In future if the process of creating the Object of B has been modified then A will gets effected, because it is instantiating, not only A there could be several class who are creating B, those all classes also will be effected.

So to hide the complex in creating the object of B from another class, let us use Factory as show below.

```
class BFactory {
  public static B createB() { // static factory method
    B b = new B();
```

```
    return b;
  }
}
class A {
  B b;
  void m1() {
    b = BFactory.createB();
  }
}
```

2. To decouple the class from the class name of another class.

```
class A {
  IB b;
  void m1() {
    b = new B();
  }
}
```

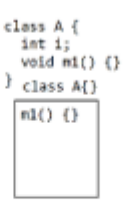
here A class has to use the class name of B to create object, if the name of the class has been changed in future from B to C, we need to modify the code inside the A and all of the classes who are instantiating the B class.

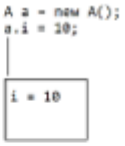
How to get the object of another class, without using name of the another class?

```
class IBFactory {
  public static IB createIB() {
    return new B();
  }
}

class A {
```

```
  IB b;
  void m1() {
    b = IBFactory.createIB();
  }
}
```

Without knowing the class name of another class, our class can get the object of another class so that we will be decoupled from the class name of another class.






```
class MessageWriter {
  private IMessageFormatter messageFormatter;

  public void writeString(String message) {
    String cMessage = null;
    messageFormatter = new HTMLMessageFormatterImpl();
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
  }
}

interface IMessageFormatter {
  String formatMessage(String message);
}

final class HTMLMessageFormatterImpl implements IMessageFormatter {
```

```
  public String formatMessage(String message) {
    return "<html><body>"+message+"</body></html>";
  }
}

final class PDFMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<pdf>"+message+"</pdf>"
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = new MessageWriter();
    messageWriter.writeMessage("Welcome to SDP");
  }
}
```

If a class is creating the object of another class, then we run into 2 problems

1. in order to create the object of another class, we need to use new operator and should pass concreate classname of another class. So that we will be tightly coupled with classname of another class

2. we will be exposed to complexity in instantiating the object of another class, so that in future if the instantiation logic has been changed then our class will be impacted.

How to solve these 2 problems?

These 2 problems are coming when our class is creating the object. So we should stop creating the object of another class in our class.

Then how does our class will get the object of another class to use the functionality?

Somebody has to create the object so that we can use it, that is where use Factory class

What is Factory class, why do we need to use it?

Factory class is used for creating the object of another class.

Why we should use Factory to create object of another class, why not our self can create the object?

There are 2 reasons for using a Factory class in creating the object

1. Factories abstracts the complexity in creating the object of another class. without knowing the details of how to instantiate the object of another class, we can use the object of another class from factory. So if the instantiation process has been changed also our class will not be affected.

2. without knowing the class name of another class, our class can get the object of another class. So that our class will be loosely coupled

```
class B {     class C {}
  C c;
  B(C c) {}
}
```

Here A wants the object of B class, how does A should get the object of B? A should not create the object of B, rather A should use Factory

```
class BFactory {
  public static B createB() {
    C c = new C();
    return new B(c);
  }
}
class A {
  B b;
  void m1() {
    b = BFactory.createB();
  }
}
```

```
class MessageFormatterFactory {
  public static IMessageFormatter createMessageFormatter(String type) {
    IMessageFormatter messageFormatter = null;
    if(type.equals("html")) {
      messageFormatter = new HTMLMessageFormatterImpl();
    }else if(type.equals("pdf")) {
      messageFormatter = new PDFMessageFormatterImpl();
    }
    return messageFormatter;
  }
}
class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    messageFormatter = MessageFormatterFactory.createMessageFormatter("html");
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
  }
}
class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    //messageFormatter = MessageFormatterFactory.createMessageFormatter("pdf");
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
  }
}
```

```
interface IMessageFormatter {
  String formatMessage(String message);
}

class HTMLMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<html><body>"+message+"</body></html>";
  }
}

class PDFMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<pdf>"+message+"</pdf>";
  }
}

class MessageFormatterFactory {
  public static IMessageFormatter createMessageFormatter(String type) {
    IMessageFormatter messageFormatter = null;
    if(type.equals("html")) {
      messageFormatter = new HTMLMessageFormatterImpl();
    }else if(type.equals("pdf")) {
      messageFormatter = new PDFMessageFormatterImpl();
    }
    return messageFormatter;
  }
}
```

```
class Test {
	public static void main(String[] args) {
		MessageWriter messageWriter = new MessageWriter();
		messageWriter.writeMessage("Welcome to Strategy Design Pattern");
	}
}
```

Here in the above, the MessageWriter will not work without using the functionality of HTML/PDFMessageFormatterImpl, so we can say our MessageWriter is dependent on these 2 classes. To the get dependent object into MessageWriter class, we are going and getting the dependent from a factory. Since we have written the logic in getting the object from some other class we are pulling dependencies from other class. So this technic of getting dependent objects by talking to some is called "Dependency Pulling".

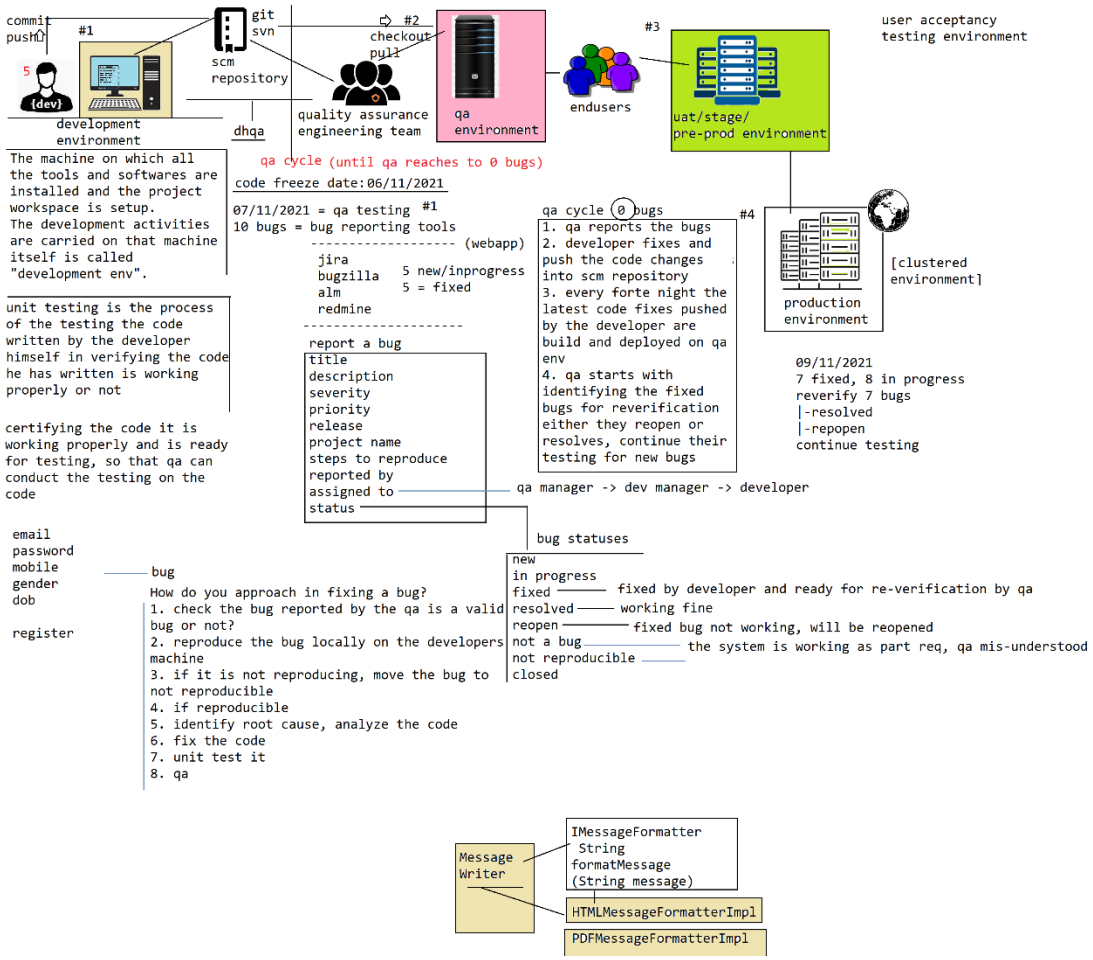
when we are pulling the dependencies by talking to someone we need to tell the logical class name of another class to get the object of another class. So that our class will become tightly coupled with logical class name of another class. So to switch from one class to another class again we need to change the logical class name.

```
class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
  }

  public void setMessageFormatter(IMessageFormatter messageFormatter) {
    this.messageFormatter = messageFormatter;
  }
}
```

```
interface IMessageFormatter {
  String formatMessage(String message);
}
class HTMLMessageFormatterImpl implements IMessageFormatter {}
class PDFMessageFormatterImpl implements IMessageFormatter {}

class  MessageFormatterFactory{
  public static IMessageFormatter createMessageFormatter(String type) {
    IMessageFormatter messageFormatter = null;
    if(type.equals("html")) {
      messageFormatter = new HTMLMessageFormatterImpl();
    }else if(type.equals("pdf")) {
      messageFormatter = new PDFMessageFormatterImpl();
    }
    return messageFormatter;
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = new MessageWriter();

    IMessageFormatter messageFormatter = MessageFormatterFactory.createMessageFormatter("html");
    messageWriter.setMessageFormatter(messageFormatter);

    messageWriter.writeMessage("welcome to sdp");
  }
}
```

In above code the amount of coupling that exists between the classes are quite minimal and can be ignorable, even then also we want to avoid the coupling and want to make classes completely loosely. To understand the reason for it let us explore how an application will have moved from development to production environment.

What are the stages in which an application will be moved from development to production env?




```
class MessageWriter {

  private IMessageFormatter messageFormatter;

  public void writeMessage(String message) {

    String cMessage = null;

    // messageFormatter = new HTMLMessageFormatterImpl();

    // messageFormatter = MessageFormatterFactory.createMessageFormatter("html");
```

```
   /**
     When we are not creating or not pull the object, we will not reference of another class inside our class, which makes our class completely loosely-coupled
   */

   cMessage = messageFormatter.formatMessage(message);
   System.out.println(cMessage);
  }
  public void setMessageFormatter(IMessageFormatter messageFormatter) {
   this.messageFormatter = messageFormatter;
  }
}

interface IMessageFormatter {
  String formatMessage(String message);
}

class HTMLMessageFormatterImpl implements MessageFormatter {
  public String formatMessage(String message) {
   return "<html><body>"+message+"</body></html>"
  }
}

class PDFMessageFormatterImpl implements MessageFormatter {
  public String formatMessage(String message) {
   return "<pdf>"+message+"</pdf>";
  }
}
```

```
class MessageFormatterFactory {
  public static IMessageFormatter createMessageFormatter(String type) {
    IMessageFormatter messageFormatter = null;

    if(type.equals("html")) {
      messageFormatter = new HTMLMessageFormatterImpl();
    }else if(type.equals("pdf")) {
      messageFormatter = new PDFMessageFormatterImpl();
    }
    return messageFormatter;
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = (MessageWriter) AppFactory.createObject("messageWriter.class");
    IMessageFormatter messageFormatter = (IMessageFormatter)
AppFactory.createObject("messageFormatter.class");

    messageWriter.setMessageFormatter(messageFormatter);
    messageWriter.writeMessage("Welcome to SDP");
  }
}
```

when we use class name or logical class name of another class inside a java class we will always be tightly coupled. So in order to switch between the classes we need to modify the source code of the java class, even though the change seems to be simple to reflect the changes we made

1. need to recompile the application
2. repackage and redeploy
3. restart the server

4. need to certify the changes by testing the application by qa

5. should follow the complete stages of delivery to release the application in production with the changes

from the above looks like a simple change in java class seems to be very costly and time taking process.

don’t write the class name or logical class name of another java class inside a java class, rather write it somewhere else, read the class name and instantiate the object

to help us in accessing the class names easily let us write the class names in properties file. properties are a key/value format file in which we can store key as identifier of the class and value as full qualified name of the class.

app-classes.properties

```
messageWriter.class=com.sdp.entities.MessageWriter
messageFormatter.class=com.sdp.entities.PDFMessageFormatterImpl
```

```
class AppFactory {
  public static Object createObject(String lclassname) {
    String fqnClass = null;

    Properties props = new Properties();
    props.load(new FileInputStream(new File("d:\\app-classes.properties")));

    fqnClass = props.getProperty(lclassname);
    Class clazz = Class.forName(fqnClass);

    Object obj = clazz.newInstance();
    return obj;
  }
}
```

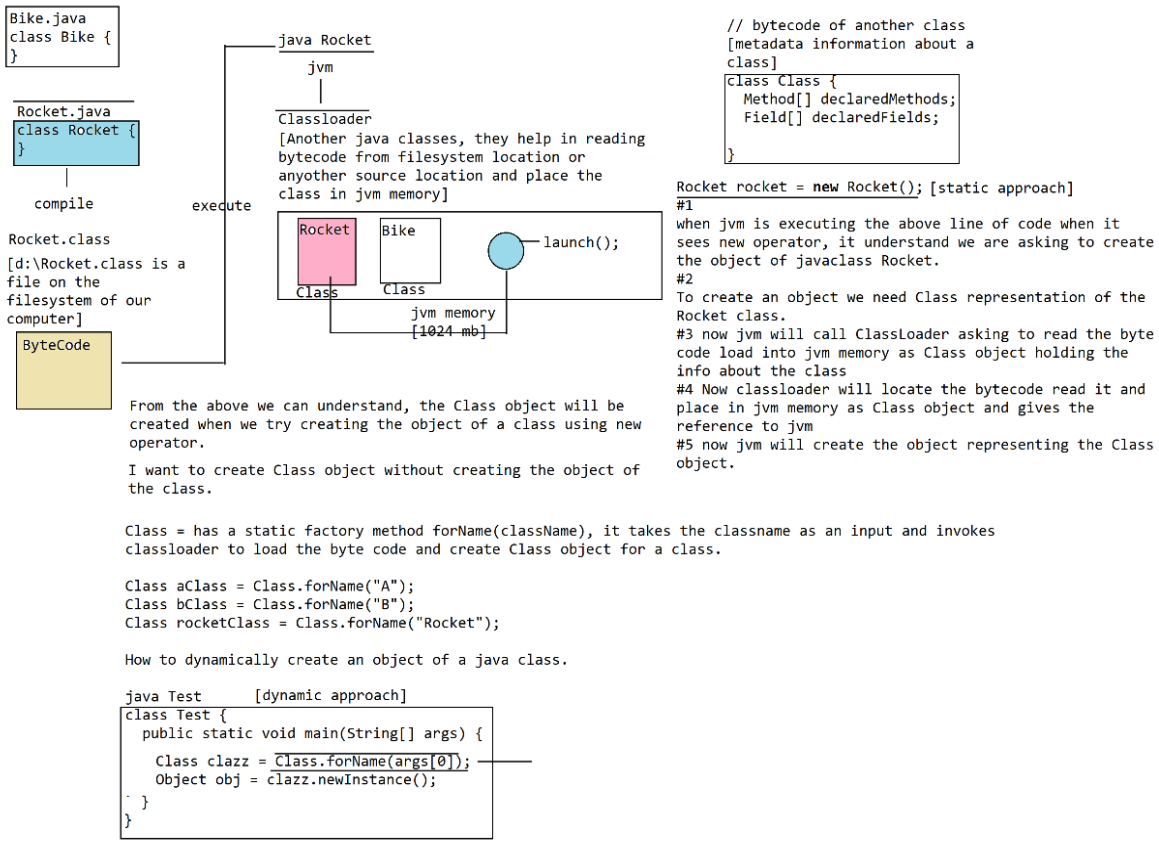
app-classes.properties

---

messageWriter.class=com.sdp.beans.MessageWriter

messageFormatter.class=com.sdp.beans.HTMLMessageFormatterImpl

```
class AppFactory {
  public static Object createObject(String lclassname) {
    String fqnClass = null;

    Properties props = new Properties();
    props.load(new FileInputStream(new File("d:\\app-classes.properties")));
    // with above line of code, all the key/value pairs in the properties file are loaded into Properties collection
    fqnClass = props.getProperty(lclassname);

    Class clazz = Class.forName(fqnClass);
    Object obj = clazz.newInstance();
    return obj;
  }
}
```



sdpattern (maven project)

|-src

|-main

|-java (java sourcecode) (compilable sourcecode) (ide/maven tool)

|-com

|-sdp

|-beans

|-MessageWriter.java

|-HTMLMessageFormatterImpl.java

|-PDFMessageFormatterImpl.java

|-helper

|-AppFactory.java

```
        |-test
          |-SDPTest.java
    |-resources (non-java sourcecode) (non-compilable sourcecode)
      |-appClasses.properties

|-pom.xml
|-target
  |-classes (hidden)
    |-*.class
    |-appClasses.properties
|-sdpattern.jar
```

Source code = anything that acts as an input during/for executing the program is called source code

app-classes.properties

```
messageWriter.class=com.sdp.beans.MessageWriter
messageFormatter.class=com.sdp.beans.PDFMessageFormatterImpl
```

```
class AppFactory {
  public static Object createObject(String lclassname) {
    Properties props = new Properties();
    props.load(new FileInputStream(new File("d:\\app-classes.properties")));

    String fqnClass = props.getProperty(lclassname);
    Class clazz = Class.forName(fqnClass);
    Object obj = clazz.newInstance();
    return obj;
  }
}
```

```
class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = (MessageWriter) AppFactory.createObject("messageWriter.class");
    IMessageFormatter messageFormatter = (IMessageFormatter)
AppFactory.createObject("messageFormatter.class");
    messageWriter.setMessageFormatter(messageFormatter);
    messageWriter.writeMessage("sdp");
  }
}
class A {
  public void m1() {
    B b = new B();
    int i = b.m2();
  }
}
class B {
  public int m3() {}
}
A.class
B.class
```

```
e:\labs:/>
sdpattern
|-src
  |-main
    |-java (java sourcecode)
      |-com
      |-sdp
```

```
        |-beans
          |-MessageWriter.java
          |-HTMLMessageFormatterImpl.java
          |-PDFMessageFormatterImpl.java
        |-test
          |-SDPTest.java
        |-helper
          |-AppFactory.java
    |-resources (non-java sourcecode)
      |-appClasses.properties
|-pom.xml
|-target
  |-classes
    |-com
      |-sdp
        |-beans
          |-MessageWriter.class
          |-HTMLMessageFormatterImpl.class
          |-PDFMessageFormatterImpl.class
        |-test
          |-SDPTest.class
        |-helper
          |-AppFactory.class
    |-appClasses.properties
```

How many types of sources codes are there?

There are 2 types of source code are there

1. java / compliable source code
2. non-java / non-compliable source code

both will act as an input for executing the program.

To help maven build-tool and ide to distinguish which or java source code and non-java source code there are 2 separate directories src/main/java (java source code), src/main/resources (non-java source code) defined by maven.

The maven or ide compiles the code under src/main/java and copies the class files into target/classes directory under the project. Similarly, if we write any non-java files in src/main/resources/ directory it will copy those files under target/classes as the non-java files are also required for executing the program.

in-short: -

when packaging the application as a jar/war/ear to deliver to the customer both java/non-java source should be included.

So maven build-tool or ide will include both the class files / non-java source code into end artifact.

```
class AppFactory {
  public static Object createObject(String lClassname) {
    Properties props = new Properties();

    InputStream in =
MessageWriter.class.getClassLoader().getResourceAsStream("appClasses.properties");
    props.load(in);

    String fqnClass = props.getProperty(lClassname);
    Class clazz = Class.forName(fqnClass);
    Object obj = clazz.newInstance();
    return obj;
  }
}
```

In the above AppFactory class we are reading the properties file pointing to the file location using absolute path. if we use the absolute path, when we move the application from one location to another location the path to the resource we are reading will be broken, again we need to modify the code in pointing to the correct location, this will put our self into maintainability issues of always modifying and redeploying or re-running the application.

Instead of using absolute path we can use relative path relative from the directory location of the classpath of our project.

Classloaders knows the resources of our project based on CLASSPATH variable we set, so we can easily ask the classloader to read the resources of our project by providing relative path location from the CLASSPATH of our project as

AppFactory.class.getClassLoader().getResourceAsStream("appClasses.properties")

ClassLoader goto the classpath location of the project and read a file appClasses.properties and create InputStream object and return.

set classpath=e:\labs\sdpattern\target\classes

java com.sdp.test.SDPTest

The resources that are internal to the application can be accessed using relative path (relative from the classpath location of the project), instead of using absolute path. only the resources that are outside the project should be accessed using absolute path.

e:\appClasses.properties (outside the project)

---

messageWriter.class=com.sdp.beans.MessageWriter

htmlMessageFormatter.class=com.sdp.beans.HTMLMessageFormatterImpl

pdfMessageFormatter.class=com.sdp.beans.PDFMessageFormatterImpl

messageBoard.class=com.sdp.beans.MessageBoard

```
d:\work\:/>
sdpattern
|-src
  |-main
    |-java
      |-com
        |-sdp
          |-beans
            |-MessageWriter.java
            |-HTMLMessageFormatterImpl.java
            |-PDFMessageFormatterImpl.java
          |-helper
            |-AppFactory.java

    |-resources
      |-appClasses.properties
|-pom.xml
|-target
  |-classes
      |-com
        |-sdp
          |-beans
            |-MessageWriter.class
            |-HTMLMessageFormatterImpl.class
            |-PDFMessageFormatterImpl.class
          |-helper
            |-AppFactory.class
```

```
class AppFactory {
  private static Properties props;

  static {
    props = new Properties();
    InputStream in =
MessageWriter.class.getClassLoader().getResourceAsStream("appClasses.properties");
    props.load(in);
  }

  public static Object createObject(lclassname){

    //props.load(new FileInputStream(new File("e:\\appClasses.properties"))); // absolute path
    String fqnClass = props.getProperty(lclassname);
    Class clazz = Class.forName(fqnClass);
    Object obj = clazz.newInstance();
    return obj;
  }
}

class Test {
  public static void main(String args[]) {
    MessageWriter messageWriter = (MessageWriter) AppFactory.createObject("messageWriter.class");
    MessageBoard messageBoard = (MessageBoard) AppFactory.createObject("messageBoard.class");

    IMessageFormatter htmlMessageFormatter = (IMessageFormatter)
AppFactory.createObject("htmlMessageFormatter.class");
    IMessageFormatter pdfMessageFormatter = (IMessageFormatter)
AppFactory.createObject("pdfMessageFormatter.class");
```

```
    messageWriter.setMessageFormatter(htmlMessageFormatter);
    messageBoard.setMessageFormatter(htmlMessageFormatter);



    messageWritter.writeMessage("welcome to sdp");
    messageBoard.flashMessage("Welcome to SDP");
  }
}



class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    cMessage = messageFormatter.convert(message);
    sop(cMessage);
  }
  public void setMessageFormatter(IMessageFormatter messageFormatter) {
    this.messageFormatter = messageFormatter;
  }
}



class MessageBoard {
  private IMessageFormatter messageFormatter;

  public void flashMessage(String message) {
    String cMessage = null;
```

```
   cMessage = messageFormatter.format(message);
   // display iton the message board
  }
  public void setMessageFormatter(IMessageFormatter messageFormatter) {
   this.messageFormatter = messageFormatter;
  }
}
```

appClasses.properties [classes and their dependencies]

---

htmlMessageFormatter.class=com.sdp.beans.HTMLMessageFormatterImpl

pdfMessageFormatter.class=com.sdp.beans.PDFMessageFormatterImpl

messageWriter.class=com.sdp.beans.MessageWriter

messageWriter.messageFormatter=htmlMessageFormatter

messageBoard.class=com.sdp.beans.MessageBoard

messageBoard.messageFormatter=htmlMessageFormatter

#1 Dont write classname of a class in another java class, it makes our classes tightly coupled

#2 Dont manage the dependencies in java class, to switch dependencies again we need to modify the code

declare classes and their dependencies in non-java files (properties). then write the code in reading the class declarations and dependency information in creating the objects of your application

We have to write complex logic in AppFactory in creating the objects for the classes we delcared and to manage their dependency to achieve loosely coupling

class AppFactory { (enhance the code)

```
  private static Properties props = new Properties();
```

```
  static {
    props.load(AppFactory.class.getClassLoader().getResourceAsStream("appClasses.properties"));
  }

  public static Object createObject(String lclassname) {
    String fqnClass = null;
    Class<?> clazz = null;
    Object obj = null;

    fqnClass = props.getProperty(lclassname);
    clazz = Class.forName(fqnClass);
    obj = clazz.newInstance();

    return obj;
  }
}

class Test {
  public static void main(String[] args) {
    MessageWriter messageWriter = (MessageWriter) AppFactory.createObject("messagerWriter.class");
    MessageBoard messageBoard = (MessageBoard) AppFactory.createObject("messageBoard.class");

    /*IMessageFormatter htmlMessageFormatter = (IMessageFormatter)
AppFactory.createObject("htmlMessageFormatter.class");
    IMessageFormatter pdfMessageFormatter = (IMessageFormatter)
AppFactory.createObject("pdfMessageFormatter.class");*/
    /*messageWriter.setMessageFormatter(htmlMessageFormatter);
    messageBoard.setMessageFormatter(pdfMessageFormatter);*/
```

```
    messageWriter.writeMessage("Welcome to SDP");
  }
}

class MessageWritter {
  private IMessageFormatter messageFormatter;
  public void setMessageFormatter(IMessageFormatter messageFormatter) {}
}

class MessageBoard {
  private IMessageFormatter messageFormatter;
  public void setMessageFormatter(IMessageFormatter messageFormatter) {}
}
```

How to make the classes loosely-coupled in an Application?

#1 use Strategy Design Patter in designing the application classes

#2 externalize the classes and their dependencies information in external configuration files

#3 write the code for reading class declarations and dependency information in instantiating the objects of the application classes.

Spring Core is all about instantiating the objects and managing the dependencies between the classes.

```
class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
  }
```

```
  public void setMessageFormatter(IMessageFormatter messageFormatter) {
    this.messageFormatter = messageFormatter;
  }
}

interface IMessageFormatter {
  String formatMessage(String message);
}

class HTMLMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<html><body>"+message+"</body></html>";
  }
}
class PDFMessageFormatterImpl implements IMessageFormatter {
  public String formatMessage(String message) {
    return "<pdf>"+message+"</pdf>";
  }
}
```

Spring Bean = If a class is instantiated and managed by the Spring then it is called Spring Bean

Declare information about our application classes in spring bean configuration file

application-context.xml

```
<beans>
  <bean id="messageWriter" class="com.sdp.beans.MessageWriter"/>
  <bean id="htmlMessageFormatter" class="com.sdp.beans.HTMLMessageFormatterImpl"/>
  <bean id="pdfMessageFormatter" class="com.sdp.beans.PDFMessageFormatterImpl"/>
</beans>
```

```
class Test {
  public static void main(String[] args) {
    BeanFactory beanFactory = new XMLBeanFactory("d:\\application-context.xml");
  }
}
```

```
class MessageWriter {
  private IMessageFormatter messageFormatter;
  public void writeMessage(String message) {
    String cMessage = null;
    cMessage = messageFormatter.formatMessage(message);
    System.out.println(cMessage);
  }
  public void setMessageFormatter(IMessageFormatter messageFormatter) {
    this.messageFormatter = messageFormatter;
  }
}

interface IMessageFormatter {
  String formatMessage(String message);
}
class HtmlMessageFormatterImpl implements IMessageFormatter {}
class PdfMessageFormatterImpl implements IMessageFormatter {}
```

application-context.xml

```
<?xml version="1.0" encoding="utf-8"?>
<beans>
  <bean id="messageWriter" class="com.sdp.beans.MessageWriter"/>
```

```
 <bean id="htmlMessageFormatter" class="com.sdp.beans.HtmlMessageFormatterImpl"/>
 <bean id="pdfMessageFormatter" class="com.sdp.beans.PdfMessageFormatterImpl"/>
</beans>
```

Bean Definition = is nothing but a bean declaration which provides the information about the java class that has to be instantiated and managed by Spring Framework

BeanFactory

Knows how to identify for a given bean id, the fully qualified name Class and takes care of instantiating the object and returns to us. The BeanFactory is an interface, for which Spring has provided several implementations to support different formats of Spring Bean Configuration.

We can define our application classes information in

1. properties
2. xml
3. annotations

Spring Framework has provided multiple implementations for BeanFactory interface each of them are responsible for reading the bean definition information from various different formats. Now programmer can easily switch from one configuration file format to another just by changing implementation of BeanFactory

BeanFactory

|-XMLBeanFactory

|-DefaultListableBeanFactory

```
BeanFactory beanFactory = new XMLBeanFactory(new ClassPathResource("application-context.xml"));
MessageWriter messageWriter = (MessageWriter) beanFactory.getBean("messageWriter");
IMessageFormatter htmlMessageFormatter = (IMessageFormatter)
beanFactory.getBean("htmlMessageFormatter");

messageWriter.setMessageFormatter(htmlMessageFormatter);
messageWriter.writeMessage("Welcome to Spring Framework");
```

A Resource can be either inside the project or outside the project

#1 Inside the project (application-context.xml) ClassPathResource

- we should use relative path
- need to use ClassLoader to read the resources from the ClassPath of the project

#2 Outside the project FileSystemResource

- we should use absolute path
- Use File I/O to read the files from the respective location directly

Resource

|-ClassPathResource [ClassLoader]

|-FileSystemResource [File I/O]

DEC 2021:

How many types of dependency injection are there?

There are 2 types of dependency injection are there

1. setter injection

The dependent object will be injected into the target class attribute via setter method.

2. constructor injection

The dependent object will be injected into target class by passing dependent as an argument to the constructor of the target class.

What is the different between setter and constructor injection?

#1. In case of constructor injection, the dependent object will be injected into target class during the time of instantiating the object of target class, by passing dependent as constructor argument.

whereas in case of setter injection the dependent will be injected into target class, after instantiating the target class object.

if we want to access the dependent within the target class constructor to perform initialization or any operation then you need to use construction injection otherwise you can go for setter injection

#2. the dependents are optional in case of setter injection

whereas mandatory incase of constructor injection

#3. circular dependencies between the classes cannot be managed via constructor injection,

setter injection can manage circular dependencies between the classes

---

Primitive dependency injection

IOC container not only supports injecting dependents as object types, we can even pass primitive value as dependent to the IOC container asking to inject into target class attribute which is called primitive dependency injection

```
class Address {
        private String addressLine1;
        private String addressLine2;
        private String city;
        // getters
        public void setCity(String city) {}
        public void setAddressLine2(String addressLine2) {}
        public void setAddressLine1(String addressLine1) {}
}
```

application-context.xml

```
<bean id="address" class="Address">
        <property name="city" value="hyderabad"/>
        <property name="addressLine2" value="2nd lane"/>
        <property name="addressLine1" value="chirag galli"/>
</bean>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
Address address = beanFactory.getBean("address", Address.class);
System.out.println(address.getAddressLine1());
```

Collection Dependency Injection

IOC container not only supports injecting object types or primitive values as dependents, it supports even injection collection class objects as dependents into target classes, which is collection dependency injection

IOC container supports 4 types of collection classes to be injected as dependents into target class

1. set
2. list
3. map
4. properties

collection dependency injection doesnt mean instantiate empty collection object and inject it as dependent into target class. we want to instantiate collection class objects with populated values/objects and inject them as dependents into target class attributes.

In the below example there is no use of injecting an empty List (l).

```
class A {
	List<String> l;
	public void setL(List<String> l) {}
}
```

application-context.xml

```
<bean id="a" class="A">
	<property name="l" ref="l"/>
</bean>
<bean id="l" class="java.util.ArrayList"/>
```

The IOC container only supports injection values into a bean definition object via setter / constructor injection only. unfortunately, none of the collection class has setter methods or constructor args to populate values into them.

since the collection classes in java doesn't have either setter/constructor we cannot make them as bean definitions with populated values. whereas collection classes are very popular being used in java programs if IOC container doesn't support injecting them with values it's a drawback in spring.

So to help us in injecting collection class objects with values as dependents the spring has provided special tags for each of the collection

```
class Product {
	int productNo;
	String productName;
	List<String> features;

	public Product(int productNo) {
		this.productNo = productNo;
	}
	public void setProductName(String productName) {}
	public void setFeatures(List<String> features) {}
}
```

application-context.xml

```
<bean id="product" class="Product">
	<constructor-arg value="393"/>
	<property name="productName" value="Samsung 32inch HD Led TV"/>
	<property name="features">
		<list>
			<value>Full HD</value>
			<value>3.1 Dolby sorround sound</value>
			<value>Andriod Smart TV</value>
			<value>Voice Remote</value>
		</list>
	</property>
</bean>
```

What is collection dependency injection?

IOC container supports injecting collection class objects with pre-populated dataas dependent into the target class attributes is called "collection dependency injection".

we can configure a collection class like an java.util.ArrayList or java.util.HashMap as bean definition and we can inject it as a dependent into the target class, but there is no use of injecting an empty collection class object as a dependent.

rather we want to instantiate collection class object with populated data and want to inject as dependent into target class. The only way IOC container supports injecting values into a class object is via setter/constructor, unfortunately none of the collection classes doesn't have either setter/constructor to configure as bean definitions and to populate values into them.

since the collections are heavily used classes we cannot populate them with data seems to be a drawback in spring framework, so ioc container/spring framework has provided special tags that helps us in creating the object of collection class by populating data, so that those objects can be injected as dependents into the target, which is called "collection dependency injection".

Summarize:

Collection dependency injection = there are special tags provided by IOC container to instantiate collection class objects by populating with data.

```
package com.cdi.beans;

class Product {

	private int productNo;

	private String productName;

	private List<String> features;


	public Product(int productNo) {}

	public void setProductName(String productName) {}

	public void setFeatures(List<String> features) {}

}
```

application-context.xml

```
<bean id="product" class="com.cdi.beans.Product">
	<constructor-arg value="9394"/>
	<property name="productName" value="Samsung 32inch HD Led Television"/>
	<property name="features">
		<list value-type="java.lang.String">
			<value>FullHD</value>
			<value>3.1 dolby stereo sound</value>
			<value>google assistance</value>
			<value>android smart</value>
		</list>
	</property>
</bean>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
Product product = beanFactory.getBean("product", Product.class);
System.out.println(product);
```

```
package com.cdi.beans;
class Staff {
	int staffNo;
	String fullname;
	int age;
	String gender;
	// setter
}
```

```
package com.cdi.beans;
class Store {
	int storeNo;
	String storeName;
	String location;
	Set<Staff> staff;
	public Store(int storeNo) {}
	// setter
}
```

application-context.xml

```
<bean id="staff1" class="Staff">
	<property name="staffNo" value="1"/>
	<property name="fullname" value="david"/>
	<property name="age" value="23"/>
	<property name="gender" value="Male"/>
</bean>
<bean id="staff2" class="Staff">
	<property name="staffNo" value="2"/>
	<property name="fullname" value="Harris"/>
	<property name="age" value="25"/>
	<property name="gender" value="Male"/>
</bean>
<bean id="staff3" class="Staff">
	<property name="staffNo" value="3"/>
	<property name="fullname" value="Kevin"/>
	<property name="age" value="25"/>
	<property name="gender" value="Male"/>
</bean>
```

```
<bean id="store" class="Store">
        <constructor-arg value="938"/>
        <property name="storeName" value="Peakwood Store"/>
        <property name="location" value="green hills"/>
        <property name="staff">
                <set value-type="com.cdi.beans.Staff">
                        <ref bean="staff1"/>
                        <ref bean="staff2"/>
                        <ref bean="staff3"/>
                </set>
        </property>
</bean>
```
What is collection dependency injection?

IOC container supports injection collection class objects with prepopulated data as dependents into the target class. to support this IOC container has introduced special tags.

There are 4 types of collections are supported

1. list

```
<list value-type="ClassType">
        <value></value> (or) <ref bean="beanId"/>
        <value></value> (or) <ref bean="beanId"/>
</list>
```

2. set

```
<set value-type="ClassType">
        <value></value> (or) <ref bean="beanId"/>
        <value></value> (or) <ref bean="beanId"/>
</set>
```

3. map

Map is a key/value pair based collection where it stores collection of entities inside it, each entry contains key/value pair.

```
class Person {
	String uidai;
	String fullname;
	int age;
	String gender;
	// setter
}

class ElectionComission {
	Map<String, Person> electrolList;
	// setter
}
```

application-context.xml

```
<bean id="person1" class="Person">
	<property name="uidai" value="u39833"/>
	<property name="fullname" value="blake"/>
	<property name="age" value="33"/>
	<property name="gender" value="Male"/>
</bean>

<bean id="person2" class="Person">
	<property name="uidai" value="u58487"/>
	<property name="fullname" value="Johnson"/>
	<property name="age" value="32"/>
	<property name="gender" value="Male"/>
</bean>
```

```
<bean id="electionComission" class="ElectionComission">
	<property name="electrolList">
		<map key-type="java.lang.String" value-type="Person">
			<entry key="v90383" value-ref="person1"/>
			<entry key="v92873" value=ref="person2"/>
		</map>
	</property>
</bean>
```

syntax: -

```
<map key-type="KEY-TYPE" value-type="VALUE-TYPE">
	<entry key="" value=""/>
	<entry key="" value=""/>
</map>
```

(or)

```
<map key-type="KEY-TYPE" value-type="VALUE-TYPE">
	<entry key-ref="id" value-ref="id"/>
</map>
```

(or)

nested tag notation: -

```
<map key-type="KEY-TYPE" value-type="VALUE-TYPE">
	<entry key="" (or) key-ref="">
		<value></value> (or) <ref bean="id"/>
	</entry>
</map>
```

#4. Properties

properties collection is also a key/value pair based collection, but the key/value both should be of String type only.

syntax: -

map -> entry [key|value] (both can be any type)

properties -> property [key|value] (both are string)

```
<props>
	<prop key="k1">
		v1
	</prop>
	<prop key="k2">
		v2
	</prop>
</props>
```

```
class Profile {
	String fullname;
	int age;
	String gender;
	private Properties wishlist;

	// setter
}
```

application-context.xml

```
<bean id="profile" class="Profile">
        <property name="fullname" value="kevin"/>
        <property name="age" value="32"/>
        <property name="gender" value="Male"/>
        <property name="wishlist">
                <props>
                        <prop key="bike">
                                Harley davidson
                        </prop>
                        <prop key="car">
                                porche
                        </prop>
                        <prop key="house">
                                Duplex Villa
                        </prop>
                </props>
        </property>
</bean>
```

The above way of working with collection dependency injection has few limitations:-

1. we cannot choose the collection implementation class we want to create for the interface

```
List<String> l = new ArrayList<>();
= new Vector<>();
= new LinkedList();
= new Queue();
```

2. we cannot define collection class object as a bean definition independently and cannot reuse across the bean definitions, we only have to define collection object inside the <property> tag or <constructor-arg> where you want to inject as shown below.

```
<bean id="a" class="A">
        <property name="l">
                <list value-type="java.lang.String">
                        <value>v1</value>
                        <value>v2</value>
                        <value>v3</value>
                </list>
        </property>
</bean>

<bean id="b" class="B">
        <property name="l">
                <list value-type="java.lang.String">
                        <value>v1</value>
                        <value>v2</value>
                        <value>v3</value>
                </list>
        </property>
</bean>
```

the above way of defining collections will duplicate list twice

To overcome the above, dis-advantages spring has provided newer tags for support collections under util namespace. instead of writing <list> <set> <map> <props> tags we need to write them from util namespace as below.

```
<util:list>
<util:set>
<util:map>
<util:props>
```

Map tag for creating map of key/value pair

```
<map key-type="java.lang.String" value-type="com.cdi.beans.Person">
        <entry key="" (or) key-ref="" value="" (or) value-ref=""/>
</map>
<map key-type="java.lang.String" value-type="com.cdi.beans.Person">
        <entry key="" or key-ref="">
                <value></value>
                or
                <ref bean=""/>
        </entry>
</map>
```

How to work with Properties collection?

```
<props>
        <prop key="">
                value
        </prop>
</props>
```

There are 2 problems are there with above collection tags

1. we cannot specify the implementation class to be instantiated for the interface collection

2. we cannot declare collection class as a bean definition and reuse it.

to overcome the above problems the ioc container has introduced util namespace. in util namespace the collection tags are same as normal collection tags only difference is the above 2.

```
class JobSheet {
        int jobSheetNo;
        String rtaNo;
        List<String> repairs;
        // setter
}
```

application-context.xml

```
<bean id="jobSheet" class="JobSheet">
  <property name="jobSheetNo" value="292"/>
        <property name="rtaNo" value="TS09JU0878"/>
        <property name="repairs" ref="repairsList"/>
</bean>

<util:list id="repairsList" list-class="java.util.LinkedList" value-type="java.lang.String">
        <value>clutch plates</value>
        <value>general servicing</value>
        <value>break pads</value>
</util:list>
```

Bean Alias

We can define more than 1 name for a bean definition, and we can access the bean from IOC container with any of the names we defined which always refers to the same bean definition.

Note: Not only the bean id even the beans names/aliases we defined for a bean should be unique across the beans of the IOC container.

How to declare alias name for a bean definition?

There are 2 ways we declare alias name for a bean definition.

1. using name attribute

2. using <alias> tag.

```
class Person {
        int personNo;
        String personName;
        int age;
        String gender;
        // setter
}
```

application-context.xml

```
<bean id="person" name="williams,goodPerson,smartPerson" class="Person">
  <property name="personNo" value="922"/>
        <property name="personName" value="Williams"/>
        <property name="age" value="32"/>
        <property name="gender" value="Male"/>
</bean>
```

The name attribute doesnt allow defining the alias name with special characters. and here "," will acts as a separator so we cannot define a alias name containing ",". To overcome the problem ioc has provided <alias> tag

```
<alias name="person" alias="sportivePerson"/>
<alias name="person" alias="badPerson"/>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("a-c.xml"));
Person person = beanFactory.getBean("smartPerson", Person.class);
```

What is bean alias?

defining more than one name for a bean definition is called "bean alias". There are 2 ways we can define alias for a bean definition.

1. name attribute
2. alias tag

```
class Key {
        private int keyNo;
        private String secret;
        // setter
}
```

application-context.xml

```
<bean id="key" name="mainkey,topkey" class="Key">
        <property name="keyNo" value="2"/>
        <property name="secret" value="0383njklho97-jh[33]"/>
</bean>
```

In name attribute the "," comma acts as an separator, so we cannot define the alias name containing "," comma inside it. to overcome this problem <alias> tag is introduced

```
<alias name="key" alias="smartkey"/>
<alias name="key" alias="hclasskey"/>
```

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("a-c.xml"));
String[] aliases = beanFactory.getAliases("topkey"); // n-1 alias names it returns
```

why do we use bean alias?

```
class OrderManagementService {
        private ParcelService bluedartParcelService;
        private ParcelService dtdcParcelService;

        public String placeOrder(List<String> products, Address address, PaymentInfo paymentInfo) {
                // create an new order with status as "pending", generates orderNo
                // add all the products to the order
                // compute the overall cost/amount of the order including discounts and taxes
                // goto payment gateway in processing the payment using payment info
                if(payment == failed) {
```

```
                // mark the ordered as failed
                // send an email to the user
                // display payment failed page
            }

            if(address.getZip() > 1000 && address.getZip() < 2000) { // cities address
                awbNo = bluedartParcelService.shipParcel(orderNo, address);
            }else {
                awbNo = dtdcParcelService.shipParcel(orderNo, address);
            }

            //String awbNo = parcelService.shipParcel(orderNo, address);
            // update order data in database with awbNo, status=accepted
            return orderNo;
        }

        public void setBluedartParcelService(ParcelService bluedartParcelService) {
            this.bluedartParcelService = bluedartParcelService;
        }
        public void setDtdcParcelService(ParcelService dtdcParcelService) {
            this.dtdcParcelService = dtdcParcelService;
        }
}

interface ParcelService {
        String shipParcel(String orderNo, Address address);
}

// headoffice = 20 days
```

```
class BluedartParcelServiceImpl implements ParcelService {
	public String shipParcel(String orderNo, Address address) {
		// store the shipment information and generate awbNo
		return awbNo;
	}
}

class DtdcParcelServiceImpl implements ParcelService {
	public String shipParcel(String orderNo, Address address) {
		return awbNo;
	}
}
```

application-context.xml

```
<bean id="bluedartParcelService" class="BluedartParcelServiceImpl"/>
<bean id="dtdcParcelService" class="DtdcParcelServiceImpl"/>

<!--<alias name="dtdcParcelService" alias="bluedartParcelService"/>-->

<bean id="orderManagementService" class="OrderManagementServiceImpl">
	<property name="bluedartParcelService" ref="bluedartParcelService"/>
	<property name="dtdcParcelService" ref="dtdcParcelService"/>
</bean>

<bean id="parcelManagementService" class="ParcelManagementService">
	<property name="bluedartParcelService" ref="bluedartParcelService"/>
	<property name="dtdcParcelService" ref="dtdcParcelService"/>
</bean>
```

Bean Autowiring

What is bean autowiring?

instead of we declaring the dependencies between the bean definitions, we want ioc container to identify the dependencies automatically and inject them, which is called "bean autowiring".

(or)

ioc container byitself will detect and manage the dependencies between the bean definitions

when we declare classes as bean definitions, without declaring the dependencies ioc container will not automatically manage them because bean autowiring is turned off by default. so to manage dependencies between the beans either we can manually declare the dependencies or enable/turnon autowiring

How to turn-on the bean autowiring?

For the bean definition we want ioc container to manage dependencies automatically, on that bean definition tag write an attribute autowire="mode" to enable/turn-on autowiring for that bean

here the mode of autowiring tells the ioc container 2 things:

1. how to identify the dependency

2. how to inject the dependency

there are 4 modes of autowiring are there

1. byName

2. byType

3. constructor

4. autodetect (deprecated in spring 3.0 and removed thereafter)

```
class SalesOrder {
        int orderNo;
        int quantity;
```

```
	double amount;

	Distributor distributor;
	public void setDistributor(Distributor distributor) {}
}

class Distributor {
	String udno;
	String distributorName;
	// setters
}
```

application-context.xml

```
<bean id="salesorder" class="SalesOrder" autowire="byName">
	<property name="orderNo" value="o9387383"/>
	<property name="quantity" value="10"/>
	<property name="amount" value="9383"/>

</bean>

<bean id="distributor1" class="Distributor">
	<property name="udno" value="ud9383"/>
	<property name="distributorName" value="Venkateswara Distributors"/>
</bean>

<bean id="godown" class="Godown"/>
<bean id="worker" class="Worker"/>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("a-c.xml"));

SalesOrder salesorder  = beanFactory.getBean("salesorder", SalesOrder.class);
```

byName = identify the dependent object matching with attribute name of the target class with the bean id in ioc container, once identified a matching bean definition inject it via setter injection

What is bean autowiring?

instead of we declaring the dependencies between the bean definitions, ioc container takes care of identifying and managing the dependencies automatically, when we use "bean autowiring".

by default bean autowiring is turned off, so in order to manage the dependencies either we need to declare dependencies information manually or enable autowiring.

To enable autowiring we need to write an attribute at the bean tag level as autowire="mode". we need to write autowire="mode" for whichever the bean we want ioc container to manage automatically on that bean tag level we need to write.

Mode of autowiring tells 2 things

1. identify the dependent bean to be injected

2. how to inject the dependency

There are 4 modes of autowiring are there

1. byName

2. byType

3. constructor

4. autodetect (deprecated and removed from 3.x)

#1. byName

identify the dependent bean matching with attribute name of the target class with bean definition id, and inject it via setter injection

if there is no bean definition id matching with attributeName, the ioc container ignores performing dependency injection as (setter injection) is optional and instantiates the object of the target and returns to us.

```
class SalesOrder {
	Distributor distributor;
	TransportType transportType;

	public SalesOrder(Distributor distributor) {
		this.distributor = distributor;
	}

	public SalesOrder(TransportType transportType) {
		this.distributor = distributor;
		this.transportType = transportType;
	}
}
```

```
class TransportType {
	String transportMode;
	String vehicleType;
	// setters
}
```

```
class Distributor {
	String udno;
	String distributorName;

	// setter
}
```

application-context.xml

```
<bean id="salesorder" class="SalesOrder" autowire="constructor">
</bean>

<bean id="distributor1" class="Distributor">
	<property name="udno" value="ud893788"/>
	<property name="distributorName" value="Raghavendra Distributors"/>
</bean>

<bean id="transportType" class="TransportType">
	<property name="tranportMode" value="road"/>
	<property name="vehicleType" value="mini van"/>
</bean>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("a-c.xml"));

SalesOrder salesOrder = beanFactory.getBean("salesorder", SalesOrder.class);
```

sop(salesOrder);

#2. byType

identify the dependenty bean definition object to be injected, by matching with attributeType of target class with bean definition class type, inject it via setter injection

if there are 2 bean definitions of matching class type with attributeType, ioc container will fail in identifying uniquely and runs into ambiguty error. we can resolve the problem by writing autowire-candidate = "false" for one of the bean definition.

autowire-candidate="false" = indicates ioc container not to consider that bean definition for autowiring.

#3. constructor

identify the dependent bean matching with constructor argument type with bean definition classType and perform the injection by passing it as an argument to the target class constructor.

in case of constructor mode, the dependent bean definition is mandatory to be injected, if the dependent is not available matching with argumentType with bean class type, the ioc container fails in creating the object of the target and throws exception

if there are more than one bean definition matching with bean classType with constructor argumentType, ioc container runs into ambiguty and fails in creating the object for the bean definition. again to resolve the problem we need to use autowire-candidate="false".

What is autowiring?

instead of we declaring the dependencies between the bean definitions, ioc container will takes care of identifying the dependencies and inject them automatically when we use autowiring.

by default autowiring is off, for which bean we want ioc container to automatically identify and manage the dependencies on that bean definition we need to write autowire="mode" to enable/turn-on autowiring.

Mode of autowiring indicates 2 things:

1. how to identify the dependencies

2. how to inject the dependencies

There are total 4 modes are there

1. byName

2. byType

3. constructor

4. autodetect (deprecated and removed > 3.x)

#1 byName

identify the dependency matching with attributeName of target class with bean id and inject the dependency via setter injection. if the bean definition with id matching with attributeName is not found, ioc container ignores the injection and proceeds in creating the object of target class (since setter injection is optional).

#2 byType

identify the dependency matching with attribute Type with the bean classType of the ioc container, and perform the dependency injection via setter.

- if there is no bean definition found matching with classType == attributeType, ignore dependency injection and proceed in creating the object of target class

- if more than one bean definition found matching with attributeType with ClassType, ioc container runs into ambiguty error

How to resolve the ambiguty error?

we need to mark one of the bean definition as autowire-candidate = "false", so that the bean definition marked will not be considered for autowiring

#3 constructor

identify the dependent bean matching with target class constructor argument type with bean classType in ioc container,if found perform the injection by passing the dependent bean as argument to the constructor of target class while creating the object of target class.

- if no dependent bean found matching above? since constructor injection is mandatory we run into Runtime exception

- if more than one bean definition found matching with the argType = bean ClassType? runs into ambiguty error

- if more than one constructor is available in the target class?

apply the below rules:

1. go with max argument constructor and identify the dependent beans matching with artType == classType, if found inject and create

2. if the bean definitions matching with argType==classType is not found for all of them, go with next max parameters constructor and apply the same process

3. if there are more than 1 constructors of the same no of arguments are there, then ioc container picks one of them randomly and checks to see the argType matching bean definition is available or not and performs injection otherwise process to the next constructor

if all of the constructors failed in matching with argType = classType then we run into exception as (constructor injection is mandatory) unless otherwise we have default constructor

---

Nested bean factories

```
class A {
        B b;
        public void setB(B b) {}
}

class B {}
```

a-c1.xml

```
<bean id="a" class="A">
        <property name="b" ref="b"/>
</bean>
```

a-c2.xml

<bean id="b" class="B"/>

BeanFactory beanFactory1 = new XmlBeanFactory(new ClassPathResource("a-c1.xml"));

BeanFactory beanFactory2 = new XmlBeanFactory(new ClassPathResource("a-c2.xml"));

Nested bean factory

What is nested bean factories?

nested bean factories is about managing the bean definitions across the ioc containers. The target bean and the dependent bean are not part of same ioc container, rather those are in 2 different ioc containers, but will we can manage the dependency between the beans by using nested bean factories.

There are 2 things we need to do to work with nested bean factories.

#1. nest one ioc container inside another one

#2. change the bean configuration to manage across the containers

#1. nest the ioc containers

while creating the ioc containers dont create them as separate or individual containers so that neither of them can talk to each other. instead nest one ioc container inside the another one.

when we nest one ioc container inside the another one, one will acts as parent and another one as child container.

The one that is being nested will become parent. The one that contains the reference of another container becomes child container.

The child container bean definitions can refer to the beans that are part of the parent container, but parent bean definitions cannot refer the beans that are in child container.

How to nest these 2 ioc containers?

In order to nest parent container inside the child, we need to first create parent container, there after while creating the child pass the reference of parent

BeanFactory parentFactory = new XmlBeanFactory(new ClassPathResource("parent-beans.xml"));

BeanFactory childFactory = new XmlBeanFactory(new ClassPathResource("child-beans.xml"), parentFactory);

#2 change in the configuration:

we need to modify the bean definition in such a way asking ioc container to look for dependent in parent ioc container.

parent-beans.xml

<bean id="b" class="B"/>

child-beans.xml

<bean id="a" class="A">

<property name="b" ref="b"/>

</bean>

<ref parent="beanId"/>= asks child ioc container to look for the bean inside parent container only.

instead of using <ref parent=""/> if we write ref="" attribute <property name="b" ref="b"/> it indicates look for the dependent bean in current ioc container, if it is not available goto the parent ioc container and inject.

```
class FuelTank {
        String fuelType;
        int capacity;
        // setter
}

class Motor {
        int serialNo;
        FuelTank fuelTank;
```

```
	// setter
}
```

parent-beans.xml

```
<bean id="fuelTank" class="FuelTank">
	<property name="fuelType" value="petrol"/>
	<property name="capacity" value="8"/>
</bean>
```

child-beans.xml

```
<bean id="motor" class="Motor">
	<property name="serialNo" value="S987933"/>
	<property name="fuelTank">
		<ref parent="fuelTank"/>
	</property>
</bean>
```

Test.java

```
BeanFactory parentFactory = new XmlBeanFactory(new ClassPathResource("parent-beans.xml"));
BeanFactory childFactory = new XmlBeanFactory(new ClassPathResource("child-beans.xml"),
parentFactory);

Motor motor = childFactory.getBean("motor", Motor.class);
sop(motor);
```

Static Factory Method Instantiation

By default when we configure a class as a bean definition, ioc container will tries to instantiate the object of a class by using Class.forName("className").newInstance() which is nothing but equal to using a new operator on a class.

But there are few classes which cannot be instantiated by using new operator for e.g..

java.util.Calendar = its basically an abstract class byitself, which cannot be instantiated by using new operator. to create the object of Calendar it has provided static factory method getInstance() in the class itself, which we need to call for creating object as shown below.

Calendar c = Calendar.getInstance();

if such classes has to be configured as bean definitions, we need to instruct ioc container asking him not to use new operator rather invoke static factory method on that class to get the object of the class, so that it place it as bean definition within ioc container using Static Factory Method Instantiation.

```
class JobScheduler {
	private String jobName;
	private Calendar scheduledTime;
	private int priority;

	// setters
}
```

application-context.xml

```
<bean id="jobScheduler" class="JobScheduler">
	<property name="jobName" value="data sorter"/>
	<property name="scheduledTime" ref="jobScheduledTime"/>
	<property name="priority" value="1"/>
</bean>

<bean id="jobScheduledTime" class="java.util.Calendar" factory-method="getInstance"/>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
JobScheduler js = beanFactory.getBean("jobScheduler", JobScheduler.class);
sop(js);
```

```
<bean id="c" class="java.util.Calendar"/>
Calendar c = new Calendar();
Calendar c = Class.forName("java.util.Calendar").newInstance();

Calendar c = Calendar.getInstance();
```

```
class A {

}

<bean id="a" class="A"/>

beanFactory.getBean("a", A.class);

Class.forName("A").newInstance(); // reflection api using runtime reference of the class we are instantiating the object
```

(both are same)

new A(); = static reference

Aware Interface

```
class Car {
	private IEngine engine;

	public void drive() {
		BeanFactory beanFactory = null;

		beanFactory = new XmlBeanFactory(new ClassPathResource("application-
context.xml"));
		engine = beanFactory.getBean("yamahaEngine");
```

```
		engine.start();
		System.out.println("driving the car");
	}
	/*
	public void setEngine(IEngine engine) {
		this.engine = engine;
	}
	*/
}

interface IEngine {
	void start();
}

class YamahaEngineImpl implements IEngine {
	public void start() {
		System.out.println("Yamaha engine started...");
	}
}

class SuzkiEngineImpl implements IEngine {
	public void start() {
		System.out.println("Suzki engine started...");
	}
}

class EngineFactory {
	public IEngine createEngine(String engineName) {
```

```
                if(engineName.equals("yamaha")) {
                        return new YamahaEngineImpl();
                }else if(engineName.equals("suzki")) {
                        return new SuzkiEngineImpl();
                }
                return null;
        }
}
```

application-context.xml

```
<bean id="car" class="Car">
        <!-- <property name="engine" ref="yamahaEngine"/> -->

</bean>

<bean id="yamahaEngine" class="YamahaEngineImpl"/>
<bean id="suzkiEngine" class="SuzkiEngineImpl"/>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
Car car = beanFactory.getBean("car", Car.class);
car.drive();
```

Aware Interface

```
class Car implements BeanFactoryAware {
	private IEngine engine;
	private String engineId;

	public Car(String engineId) {
		this.engineId = engineId;
	}

	public void drive() {

		engine = beanFactory.getBean(engineId, IEngine.class);

		engine.start();
		System.out.println("driving car");
	}

	public void setBeanFactory(BeanFactory beanFactory) {

	}
```

```
}

interface IEngine {
	void start();
}

class YamahaEngineImpl implements IEngine {
	void start() {
		System.out.println("Yamaha engine started");
	}
}
class SuzkiEngineImpl implements IEngine {
	void start() {
		System.out.println("Suzki engine started");
	}
}

class EngineFactory {
	public static IEngine createEngine(String manufacturer) {
		IEngine engine = null;

		if(manufacturer.equals("yamaha")) {
			engine = new YamahaEngineImpl();
		}else {
			engine = new SuzkiEngineImpl();
		}
		return engine;
	}
}
```

application-context.xml

```
<bean id="car" class="Car">
        <constructor-arg value="suzkiEngine"/>
</bean>

<bean id="yamahaEngine" class="YamahaEngineImpl"/>
<bean id="suzkiEngine" class="SuzkiEngineImpl"/>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
Car car = beanFactory.getBean("car", Car.class);

car.drive();
```

```
class Car implements BeanFactoryAware {
        private BeanFactory beanFactory;
        public void setBeanFactory(BeanFactory beanFactory) {

        }
}
```

a-c.xml

```
<bean id="car" class="Car">

</bean>
<bean id="yamahaEngine" class="YahamaEngineImpl"/>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("a-c.xml"));
Car car = beanFactory.getBean("car", Car.class);
```

Instance Factory Method Instantiation

---

nse/bse stock exchange

---------- mumbai (dalal street)

national stock exchange (nse)

bombay stock exchange (bse)

- stock exchange office = town/city all over the india = membership

individual =

        1 lac = 20 = 5 (5 companies) - portifolio

---------- stock exchange brokers / agents

1 lacs = 3 months = 3 lacs = hardly impossible = comission = 2.5% (2500/-) = fraud

---

stock exchange online comes into picture

stock trading websites (demat accounts)

- icicidirect
- sharekhan
- indiabulls
- upstock
- zerodha
- grow

nse

bse

---------- stock exchange offices ----------

brokers / agents = comission (buying and selling)

---

through the brokers there is a high chance of fraud in recommending the wrong stocks or promoting while buying/selling with false prices etc

online stock trading websites are introduced to add transperancy in trading stocks. There are lot of financial and banking companies are there to whom they provided brokerage licenses, asking to provide online trading services to the customers onbehalf of bse/nse stock exchange.

- sharekhan
- icicidirect
- zerodha
- upstox
- groww
- indiabulls

open an demat account (link with savings account), you need to give power of attorney to the borkerage vendors. transfer money from savings to demat and viceversa while buying/selling the stocks.

1. transperancy in seeing the latest stock updates
2. plenty of information about a stock will be available in the trading website. 32 weeks lowest, 32 weeks highest, volumes of trading, todays price, 7 days price
3. information about the stock listing company, their returns, invenstment, future plans.
4. daily technical calls and recommendations

They provide lot of tools in trading the stocks

1. price of stock

2. portfolio management

3. history

4. buy requests

5. sell requests

6. stop loss

the online brokerage vendors gets comissions for each buy/sell of stocks

---

How does bse/nse stock exchange employees are going to provide business services to the public?

The information about the stocks/companies are stored in the database, if a vendor/customer comes for buying or selling stock to the agents/employees of the bse stock exchange, performing the operation manually leads to lot of problems as described below:

1. every employee/agent working in bse/nse stock exchange should be an IT Engineer and has to have good knowledge in accessing/manipulating the data on the underlying database, which is high impossible.

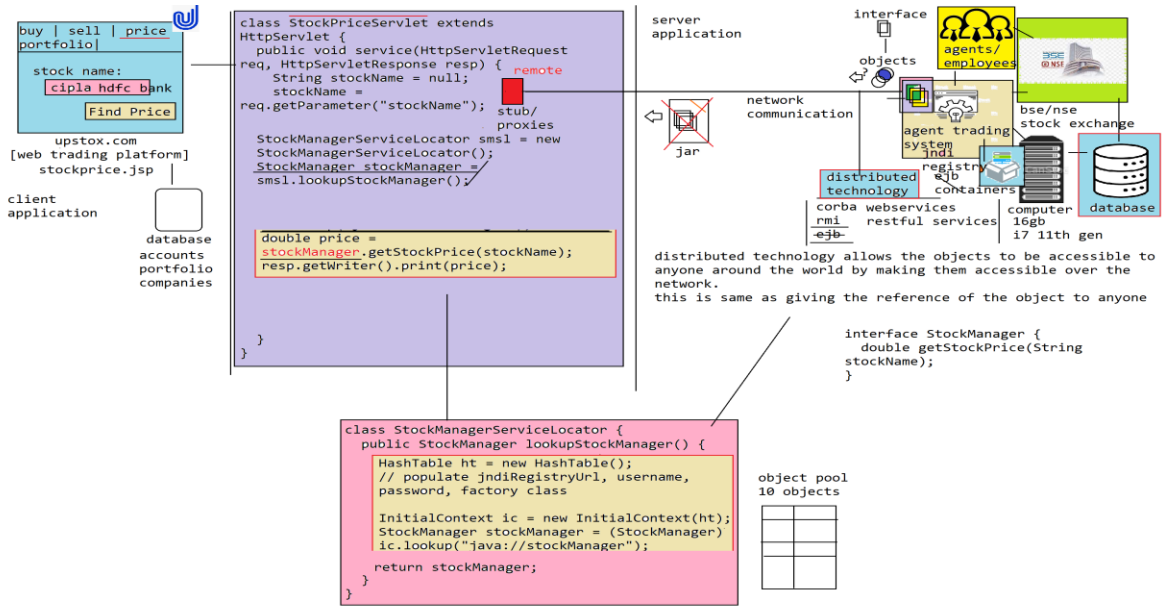
2. every request in buying/selling stocks requires an through computation operation/business calculatiion like comission charges, govt taxes etc. manually performing these calculations in alloting or selling the stocks will takes

    2.1 lot of time for the agents at bse/nse

    2.2 always there is a chance of human errors in performing these calculations which results in in-accuracy

3. while buying or selling the stocks, the agents has to execute queries on the database tables in modifying the data in allocating or releasing the stocks. its not on one table, may be they need to modify the data on bunch of tables. directly constructing sql queries in modifying the data on the underlying tables is always a high risk.

there is chance where something might go wrong while constructing the query due to which incorrectly the data would gets modified on the underlying database due to which a huge loss will incurred.



any changes on the remote application object is being protected and shielded from our application components through the help of Service Locator

How does the bse/nse stock exchange employees/agents by themself provide trading services to the stock traders?

they cannot directly go and perform operations on the underlying database in offering the trading services to the customers, because there are lot of problems in manually performing the operations on the underlying database.

1. the bse/nse stock exchange agents/employees must of the sql engineers or it engineers to carry out the business operations which is quite impossible

2. agents have to perform business calculations in providing the services if these business computations are carried manually.

2.1 it takes lot of time in performing the operations

2.2 accuracy is not guaranteed, because of human errors

3. the bse/nse stock exchange employees has to construct manually the sql queries in order to perform the operation, where these queries are not validated and their high chance where while executing the queries the data on the underlying database may be modified wrongly or corrupted due to incorrect query/conditions

to overcome the above problems, let the bse/nse exchanges built a software application with adequate functionality and provide it to the agents or employees of the exchanges.

There are lot of advantages in providing an access to application in offering business services rather than perform the operations directly on the underlying database.

1. agent/employee need not be an IT engineer

2. as the business computations are taken care by the application itself

2.1 time required to perform the business computations are very very less

2.2 there is no chance of human errors and accuracy will be very high

3. no chance of data corruption or incorrectly modifying the data, as the software application has been pre-tested and provided to the agents.

If we write the lookup logic in getting the remote reference of the distributed object across all the classes of our application, then the code gets duplicated across all the classes of our application which will put us in to several problems

1. The lookup logic we are writing here is environment specific lookup logic, if the remote application has been moved from one env to another env again we need to modify the lookup logic in pulling the remote object from a different env (ip, port etc) which requires changes across all the classes whereever we wrote the lookup logic

2. lookup logic we wrote is platform specific lookup logic if the underlying distributed object has been moved from one platform to the another, then we need modify the lookup logic jndiURL, factory class and few more properties we populate in looking up the object in all the classes of our application

3. The lookup logic we are writing is technology specific lookup logic if we switch the distributed object from one technology to another for eg.. from ejb to rmi, then the entire lookup logic we have written should be scrapped out and should rewrite again from scratch in all the classes of our application

since we have duplicated the lookup logic across all the classes of our application any change in distributed object will affect all of the components of our application.

From the upstox pointing of view the bse/nse stock exchange exposed distributed components becomes services since these components are provided functionality to the external application components.

1. where do we write the ServiceLocator class, is it on client-side application or server-side application?

2. what is the difference between ServiceLocator and Factory class?

Instance Factory Method Instantiation

#BSE/NSE Stock Exchange

AgentTradingSystem

```
interface StockTradeManager {
        double getStockPrice(String stockName);
}
// assumption: ejb technology
class BSEStockTradeManager implements StockTradeManager {
        double getStockPrice(String stockName) {
```

```
                // goto database, execute query
                // retreive price of the stock
                return 1093.3;
        }
}

#upstox.com
------------
interface StockTradeManager {
        double getStockPrice(String stockName);
}

class StockPriceController {
        private StockTradeManager stockTradeManager;

        public double findStockPrice(String stockName) {
                double price = stockTradeManager.getStockPrice(stockName);
                return price;
        }

        public void setStockTradeManager(StockTradeManager stockTradeManager) {
                this.stockTradeManager = stockTradeManager;
        }
}

class AgentTradingSystemServiceLocator {
        public Object lookupAgentTradingSystemService(String jndiName) {
                HashTable ht = new HashTable();
                ht.put("jndiFactoryUrl", "http://192.293.29.29:9990/");
```

```
                ht.put("username", "weblogic");
                ht.put("password", "welcome1");
                ht.put("jndiFactoryClass", "com.weblogic.jndi.factory.WeblogicJndiFactoryImpl");


                InitialContext ic = new InitialContext(ht);
                // stub/proxy
                Object remoteObject = ic.lookup(jndiName);
                return remoteObject;
        }
}
```

application-context.xml

```
<bean id="stockPriceController" class="StockPriceController">
        <property name="stockTradeManager" ref="stockTradeManager"/>
</bean>
<bean id="agentTradingSystemServiceLocator" class="AgentTradingSystemServiceLocator"/>

<bean id="stockTradeManager" factory-bean="agentTradingSystemServiceLocator" factory-
method="lookupAgentTradingSystemService">
        <constructor-arg value="java://bseStockTradeManager"/>
</bean>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
StockPriceController spc = beanFactory.getBean("stockPriceController", StockPriceController.class);
double price = spc.findStockPrice("cipla");
```

System.out.println("price : " + price);

What is instance factory method instantiation?

by default when configure a class as bean definition the ioc container will creates the object of the class using new operator, but there are some classes for whom we cannot instantiate the object directly using new operator inorder to instantiate the object we have to create the object of another class and call factory method on the object to create object of another class.

For eg.. when we are working the distributed technology objects like ejb or rmi etc we write the code for pulling the remote references of the objects in a Service Locator class. Now in order to instantiate or pull the remote references of ejb/rmi and place it as bean definitions we need to ask ioc container instantiate the object of ServiceLocator and invoke the lookupMethod on it which returns remote object place it as an bean definition within ioc container.

The above technic of placing the objects in ioc container is called "instance factory method instantiation".

if ioc container doesnt support instance factory method instantiation, the object way of such objects to be used within our class application class is through dependency pull which makes our application classes tightly coupled.

BSE/NSE Exchange

Agent Trading System Application

```
interface StockTradeManager {
        double getStockPrice(String stockName);
}

// assume: ejb
class BSEStockTradeManagerImpl implements StockTradeManager {
        double getStockPrice(String stockName) {
                return 938;
```

```
	}
}
```

Upstox.com

```
class GetStockPriceController {
	private StockTradeManager stockTradeManager;

	public double findStockPrice(String stockName) {
		/**
		// dont write dependency pulling logic, because we are tightly coupled through logical class name, inject StockTradeManager as dependency into our class

		AgentTradingSystemServiceLocator atssl = new AgentTradingSystemServiceLocator();

		stockTradeManager = (StockTradeManager) atssl.lookupAgentTradingSystemService("java://bseStockTadeManager");
		*/
		double price = stockTradeManager.getStockPrice(stockName);
		return price;
	}
	public void setStockTradeManager(StockTradeManager stockTradeManager) {
		this.stockTradeManager = stockTradeManager;
	}
}

class AgentTradingSystemServiceLocator {
	public Object lookupAgentTradingService(String jndiName) {
```

```
                Object remote = null;
                HashTable ht = new HashTable();
                ht.put("jndiUrl", "t3://localhost:9091");
                ht.put("username", "weblogic");
                ht.put("password", "welcome1");
                ht.put("jndiFactoryClass", "com.weblogic.jndi.WeblogicJNDIFactory");

                InitialContext ic = new InitialContext(ht);
                remote = ic.lookup(jndiName);
                return remote;
        }
}
```

application-context.xml

```
<bean id="getStockPriceController" class="GetStockPriceController">
        <property name="stockTradeManager" ref="stockTradeManager"/>
</bean>

<bean id="agentTradingSystemServiceLocator" class="AgentTradingSystemServiceLocator"/>

<bean id="stockTradeManager" factory-bean="agentTradingSystemServiceLocator" factory-
method="lookupAgentTradingService">
        <constructor-arg value="java://bseStockTradeManager"/>
</bean>
```

Test.java

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("a-c.xml"));

GetStockPriceController gspc = beanFactory.getBean("getStockPriceController",
GetStockPriceController.class);

double price = gspc.getStockPrice("cipla");

System.out.println("price :"  +price);
```

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("application-context.xml"));
```

In Spring 4.x onwards the XMLBeanFactory class has been deprecated and in Spring 5 version it was removed.

BeanFactory (interface)

|-DefaultListableBeanFactory = is an implementation in which we can list down all the bean definitions in it

Spring 5.x

How to create ioc container?

```
DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

beanDefinitionReader.loadBeanDefinitions(new ClassPathResource("application-context.xml"));

PropertiesBeanDefinitionReader propsBeanDefinitionReader = new
PropertiesBeanDefinitionReader(beanFactory);

propsBeanDefinitionReader.loadBeanDefinitions(new ClassPathResource("application.properties"));
```