```
===================
Spring Core module
===================

=> Base module of Spring Framework

=> Providing fundamental concepts of spring framework

                1) IOC
                2) DI
                3) Autowiring

###### Spring Core module is used to manage our classes in the project. ######


=> In a project we will have several classes

                        1) Controller classes (handle request & response)

                        2) Service classes (handle business logic)

                        3) DAO classes (handle DB ops)

=> In project execution process, One java class should call another java class method

Ex :
                1) Controller class method should call service class method

                2) Service class method should call DAO class method

=> We have 2 options to access one java class method in another java class

                1) Inheritence (IS-A)

                2) Composition (HAS-A)

===============
IS-A Relation
===============

=> Extend the properties from one class to another class.

=> Super class methods we can access directley in sub class.

Ex : Car and Engine

                Car class ----> drive ( ) method

                Engine class ---> start ( ) method

Note: If we want to drive the car then we need to start the Engine first. That means Car class
functionality is depending on Engine class functionality.

=> Car class method should call Engine class method.

------------------------------------------
package in.ashokit;

public class Engine {

        public boolean start() {
                // logic
                return true;
        }

}
---------------------------------------
package in.ashokit;

public class Car extends Engine {

        public void drive() {

                boolean status = super.start();
                if (status) {
                        System.out.println("Engine started...");
                        System.out.println("Journey started...");
                } else {
                        System.out.println("Engine having trouble...");
                }
        }
}
--------------------------------------------------------------
```

=> In the above approach car is extending properties from Engine class.

=> In future car can't extend props from other classes bcz java doesn't support multiple inheritence.

=> With IS-A relationship our classes will become tightly coupled.

=> To overcome problems of IS-A relation we can use HAS-A relation.

================
HAS-A relation
================

=> Create object and call the method

=> Inside car class create object for Engine class and call eng class method.

```
-----------------------------------------------------
public class Car {

        public void drive() {

                Engine eng = new Engine();

                boolean status = eng.start();
                if (status) {
                        System.out.println("Engine started...");
                        System.out.println("Journey started...");
                } else {
                        System.out.println("Engine having trouble...");
                }
        }
}
------------------------------------------------------------------------------
```

=> If someone modify Engine class constructor then Car class will fail...

=> with HAS-A relation also our java classes becoming tightly coupled.

Note: Always we need to develop our classes with loosely coupling.

=> To make our classes loosely coupled, we should not extend properties and we should not create object directley.

=> To make our classes loosely coupled we can use Spring Core Module concepts

```
                1) IOC Container

                2) Dependency Injection.

======================
What is IOC Container
======================

=> IOC stands for Inversion of control.

=> IOC is a principle which is used to manage & colloborate the classes and objects available in the
application.

=> IOC will perform Dependency Injection in our application.

=> Injecting Dependent class object into target class object is called as Dependency Injection.

=> By using IOC and DI we can achieve Loosely coupling among the classes in our application.

Note: We need to provide input for IOC regarding our target classes and dependent classes to perform
Dependency Injection.

Note: We can do configuration in 3 ways

        1) xml (outdated -> springboot will not support)

        2) Java based

        3) Annotations

=> IOC will take our normal java classes as input and it provides Spring Beans as output.

=====================
What is Spring Bean
=====================

=> The java class which is managed by IOC is called as Spring bean.

===============================================
First App development using Spring framework
===============================================

## Step-1 : Create maven project using IDE (Eclipse/ STS / IntelliJ)

        - select simple project (standalone)
        - groupId : in.ashokit
        - artifactId : 01-Spring-App

## Step-2 : Configure Spring dependency in project pom.xml file to download required libraries.

                URL : https://mvnrepository.com/

-------------------------------------
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>6.2.5</version>
                </dependency>
        </dependencies>
-------------------------------------

## Step-3 :: Create Required java classes
```

```
-------------------------------
package in.ashokit;

public class Engine {

        public Engine() {
                System.out.println("Engine Constructor :: Executed");
        }
}
--------------------------------
```

Step-4 :: Create Spring Bean Configuration file and configure java classes as spring beans.

        File Location : src/main/resources/spring-beans.xml

```
---------------------------------------------------------

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="e" class="in.ashokit.Engine"/>

</beans>
----------------------------------------------------------
```

Step-5 :: Create Main class to test our application.

```
-----------------------------------------------------------
public class Main {

        public static void main(String[] args) {

                // start IOC by giving xml file as input

                ApplicationContext ctxt = new ClassPathXmlApplicationContext("spring-beans.xml");

                // getting bean obj
                Engine e1 = ctxt.getBean(Engine.class);
                System.out.println(e1.hashCode());
        }
}
------------------------------------------------------------
```

=============================
What is Dependency Injection
=============================

=> The process of injecting one class object into another class object is called as dependency injection.

Note: IOC is responsible to perform dependency injection.

=> We can perform Dependency Injection in 3 ways

1) Constructor Injection

2) Setter Injection

3) Field Injection

```
===================================
What is Constructor Injection ?
===================================

=> Injecting dependent obj into target obj using target class parameterized constructor is called
Constructor injection (C.I).

// param constructor with dependent obj as constructor arg.
public Car(Engine eng) {
        this.eng = eng;
}

Note: To represent constructor injection we will use below syntax

Syntax : <constructor-arg name="" ref=""/>

<bean id="c" class="in.ashokit.Car">
        <constructor-arg name="eng" ref="e"/>
</bean>

<bean id="e" class="in.ashokit.Engine" />

===================================
What is Setter Injection ?
===================================

=> Injecting dependent obj into target obj using target class setter method is called as setter
injection (S.I).

// SETTER METHOD with dependent obj as parameter
public void setEng(Engine eng) {
        this.eng = eng;
}

Note: To represent setter injection we will use below syntax

Syntax : <property name="" ref=""/>

<bean id="c" class="in.ashokit.Car">
        <property name="eng" ref="e"/>
</bean>

<bean id="e" class="in.ashokit.Engine" />


========================
IOC with DI Example
=========================

Requirement : When we withdraw amount from ATM then it should print reciept using Printer.

Note:  Create Printer and ATM classes and manage them using IOC and DI.

--------------------------------------------------------------
package in.ashokit;

public class Printer {

        public Printer() {
                System.out.println("Printer :: 0-Param Constructor");
        }

        public void print() {
                System.out.println("Printing Reciept....");
        }
}
-------------------------------------------------------
package in.ashokit;

public class ATM {

	private Printer printer;

	public ATM() {
		System.out.println("ATM :: 0-Param Constructor");
	}

	// used for constructor injection
	public ATM(Printer printer) {
		System.out.println("ATM :: Param Constructor");
		this.printer = printer;
	}

	// used for setter injection
	public void setPrinter(Printer printer) {
		System.out.println("ATM :: setPrinter() method");
		this.printer = printer;
	}

	public void withdraw() {
		System.out.println("Amount withdrawn successfully");
		printer.print();
	}
}
-------------------------------------------------------
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="p1" class="in.ashokit.Printer"/>

    <bean id="atm" class="in.ashokit.ATM">
        <property name="printer" ref="p1"/>
        <constructor-arg name="printer" ref="p1"/>
    </bean>

</beans>
-------------------------------------------------------
package in.ashokit;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

	public static void main(String[] args) {

		ApplicationContext ctxt = new ClassPathXmlApplicationContext("beans.xml");

		ATM atm = ctxt.getBean(ATM.class);

		atm.withdraw();
	}
}
-------------------------------------------------------
```

=> In the above aplication we are not creating objects directley, but we are referring one class in another class directley.

Ex: ATM class having reference of Printer class.

=> If one class is referring another class then those classes are tightly coupled.

==
How to make our classes completley loosely coupled ?
===

=> By following Strategy Design Pattern we can make our classes loosely coupled.

=> The Strategy Design Pattern is a behavioral design pattern that enables selecting an algorithm's behavior at runtime.

=> Strategy Design pattern suggesting to follow below 3 principles while developing classes.

1) Favour composition over inheritence

2) Always code to interfaces instead of impl classes

3) Code should be open for extension and should be closed for modification

=======================
ShoppingCart Example
=======================

```
---------------------------------------------------------------------------
public interface IPayment {

        public boolean pay(double amt);

}
---------------------------------------------------------------------------
public class CreditCardPayment implements IPayment {

        public CreditCardPayment() {
                System.out.println("CreditCardPayment :: Constructor");
        }

        @Override
        public boolean pay(double amt) {
                System.out.println("CreditCard payment success...");
                return true;
        }

}
---------------------------------------------------------------------------
public class DebitCardPayment implements IPayment {

        public DebitCardPayment() {
                System.out.println("DebitCardPayment :: Constructor");
        }

        @Override
        public boolean pay(double amt) {
                System.out.println("DebitCard payment success...");
                return true;
        }

}
---------------------------------------------------------------------------
public class ShoppingCart {
	private IPayment payment;

	public ShoppingCart() {
		System.out.println("ShoppingCart :: 0-Param Constructor");
	}

	public ShoppingCart(IPayment payment) {
		System.out.println("ShoppingCart :: Param Constructor");
		this.payment = payment;
	}

	public void setPayment(IPayment payment) {
		System.out.println("setPayment() - called...");
		this.payment = payment;
	}

	public void checkout() {

		boolean status = payment.pay(1000.00);

		if (status) {
			System.out.println("Order placed succcessfully..");
		} else {
			System.out.println("Payment failed...");
		}
	}
}
---------------------------------------------------------------------------
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="cp" class="in.ashokit.CreditCardPayment" scope="prototype"/>

	<bean id="dp" class="in.ashokit.DebitCardPayment" scope="prototype"/>

	<bean id="sc" class="in.ashokit.ShoppingCart" scope="prototype">
		<constructor-arg ref="cp" />
	</bean>

</beans>
---------------------------------------------------------------------------
public class TestApp {

	public static void main(String[] args) {

		ApplicationContext ctxt = new ClassPathXmlApplicationContext("beans.xml");

		ShoppingCart sc1 = ctxt.getBean(ShoppingCart.class);
		sc1.checkout();
	}
}
```

=============
Bean Scopes
=============

=> Bean scope represents how many objects should be created for spring bean by IOC container.

=> We have below bean scopes in spring

```
        1) Singleton (default)

        2) Prototype

        3) Request

        4) Session

-------------
Singleton
-------------

=> singleton is default scope.

=> Only one instance will be created for spring bean.

=> Singleton scoped beans objects will be created when IOC container started.

=> Singleton beans will follow Eager Loading.

-------------
Prototype
-------------

=> Every time new object will be created for spring bean on demand basis.

=> When we call getBean() method then only obj will be created.

=> Prototype beans will follow lazy loading.


------------------
request & session
------------------

=> These 2 scopes are belongs to spring web mvc module.


===========
Autowiring
===========

<bean id="cp" class="in.ashokit.CreditCardPayment" scope="prototype"/>

<bean id="sc" class="in.ashokit.ShoppingCart" scope="prototype">
        <constructor-arg ref="cp" />
</bean>

=> If we use 'ref' attribute to perform DI then it is called as Manual wiring.

=> If we use 'Auto Wiring' concept then IoC itself will identify dependent bean object and inject
into target object.

####
Autowiring is a feature in the Spring Framework that automatically injects the required beans
(objects) into your class without explicitly specifying them in the configuration.
####

=> To perform autowiring we have to enable it by using autowiring modes.

                1) byName
                2) byType
                3) constructor
                4) none
```

```
=======
byName
=======

=> Injects bean by matching the property name.

=> If any bean id or name matching with target bean variable name, then ioc will consider that as
dependent bean and ioc will inject that bean into target.

Note: As per below configuration IOC will identify CreditCardPayment bean name is matching with
target bean variable hence it will be injected into target bean.


<bean id="payment" class="in.ashokit.CreditCardPayment" />

<bean id="dp" class="in.ashokit.DebitCardPayment" />

<bean id="sc" class="in.ashokit.ShoppingCart" autowire="byName">

</bean>

Note : We can't configure two beans with same id hence ambiguity is not possible here.

=======
byType
=======

=> It will identify dependent bean based on type of variable available in target bean.

=> If variable data type is a class, then it will inject that class obj as dependent.

=> If variable data type is interface, then it will identify impl class objs of that interface as
dependents.

=> If we have more than one impl class for that interfce then IOC will run into ambiguity problem.

Note: To resolve byType ambiguity problem we will use "primary=true" for one bean.


<bean id="cp" class="in.ashokit.CreditCardPayment" primary="true" />

<bean id="dp" class="in.ashokit.DebitCardPayment" />

<bean id="sc" class="in.ashokit.ShoppingCart" autowire="byType">

</bean>


============
constructor
============

=> It is used to enable constructor injection using autowiring.

=> constructor mode internally uses byType to identify dependenty object.


<bean id="cp" class="in.ashokit.CreditCardPayment" primary="true" />

<bean id="dp" class="in.ashokit.DebitCardPayment" />

<bean id="sc" class="in.ashokit.ShoppingCart" autowire="constructor">

</bean>
```

```
======================================

1) What is Framework

2) Why to use frameworks

3) Spring Introduction

4) Spring Architecture

5) Spring Modules Overview

6) Spring Core

7) IOC Container

8) Dependency Injection

9) Constructor Injection

10) Setter Injection

11) Bean Scopes

12) Autowiring (byName, byType, constructor)

=========================================
```