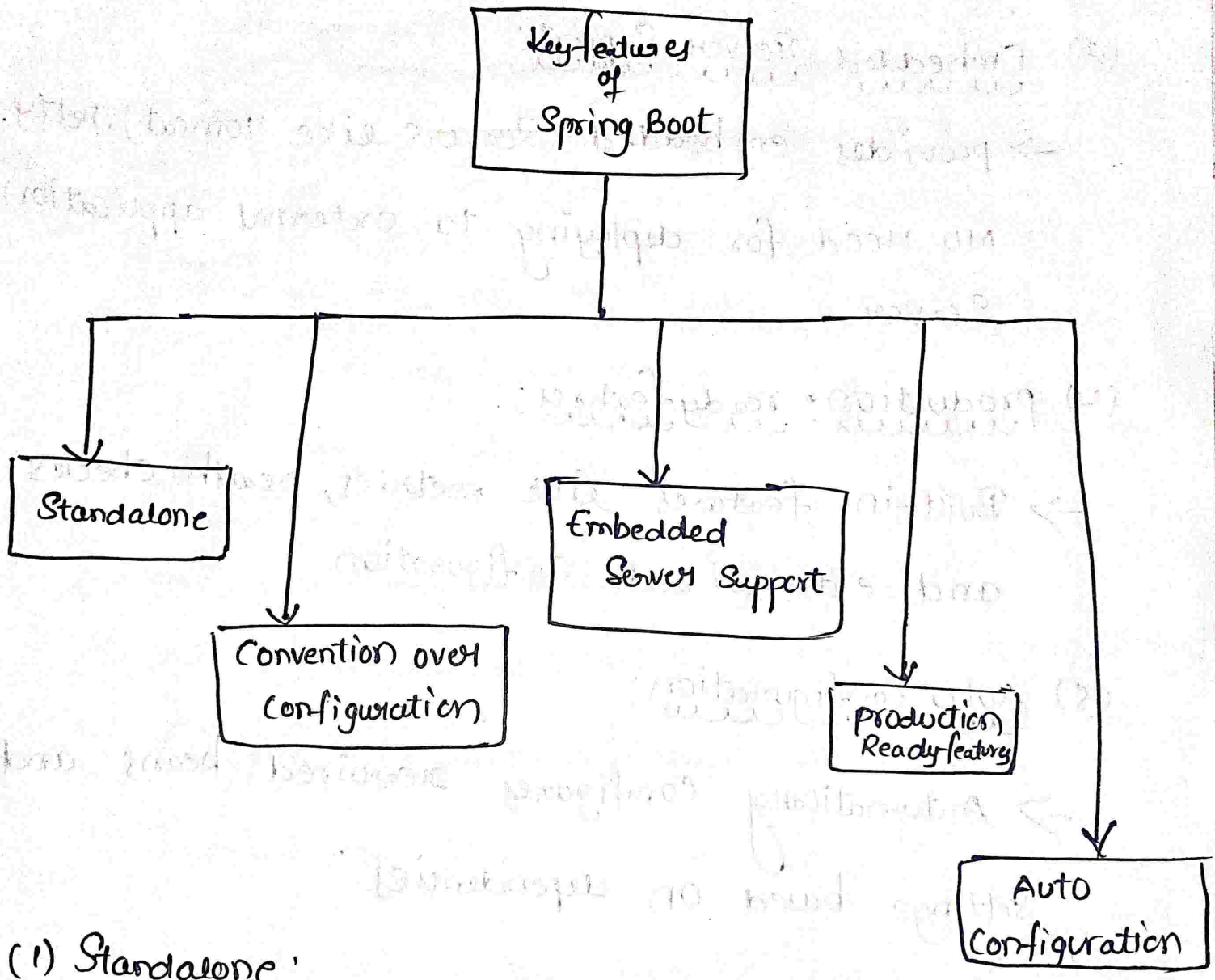
* Spring Boot Application :

Key features of Spring Boot Application



(1) Standalone :

→ Run directly using 'java -jar' NO external Server needed (Comes with embedded Servers like Tomcat / Jetty).

(2) Convention over Configuration:

→ Most configurations are handled automatically by SpringBoot, reducing boilerplate code.

(3) Embedded Server Support:

→ provides embedded Servers like Tomcat, Jetty.

No Need for deploying to external application Server.

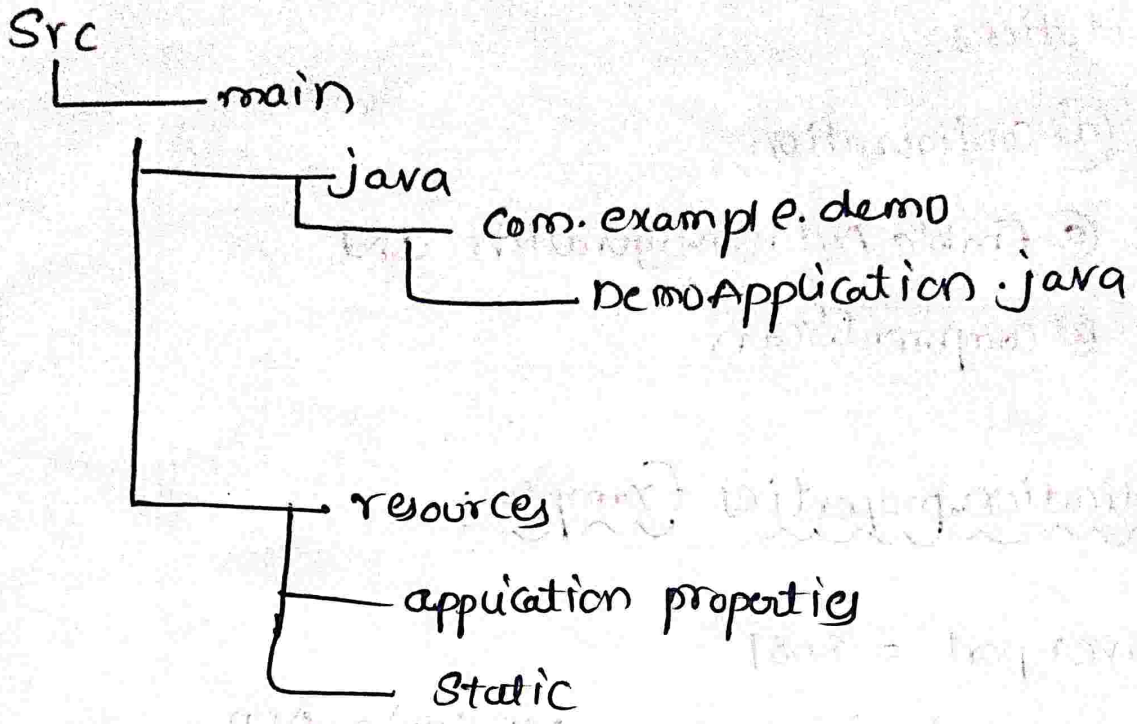
(4) production-ready features:

→ Built-in features like metrics, health checks and externalized configuration.

(5) Auto Configuration:

→ Automatically configures required beans and Settings based on dependencies.

Spring Boot Application Structure:



Example:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // main annotation
public class DemoApplication
{
    psvm(String args[])
{
SpringApplication.run(DemoApplication.class, args);
    }
}
```

Note:

@SpringBootApplication is a combination of three annotations:

→ @Configuration,

→ @EnableAutoConfiguration and

→ @ComponentScan

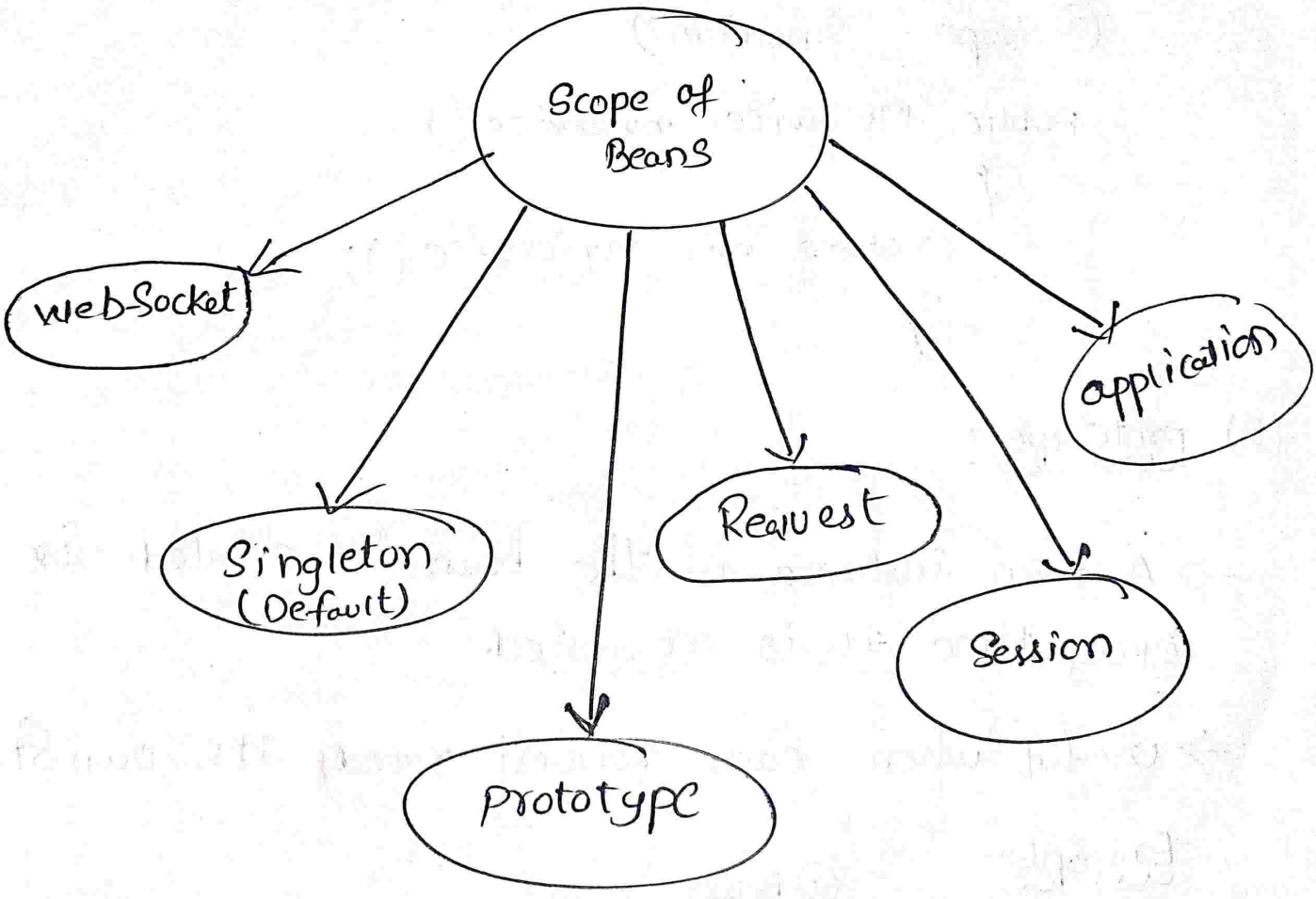
application.properties Example:

```
Server.port = 8081
Spring.application.name = MySpringApp
```

* Scope of bean :

→ In Spring Framework, the Scope of bean defines how long a bean instance will live and how many instances of that bean spring will create and manage in the container.

Scope of Beans :



(1) Singleton (Default)

→ Only one instance of the bean is created for the entire Spring Container.

→ The same instance is shared everywhere it is injected.

Example:

```
@Bean
@Scope ("Singleton")
public MyService myservice()
    {
        return new MyService();
    }
```

(2) prototype:

→ A new instance of the bean is created ~~for~~ every time it is requested.

→ useful when each request needs its own State.

Example:

```
@Bean
@Scope ("prototype")
public MyService myservice()
    {
        return new MyService();
    }
```

(3) request:

→ A new Bean instance is created for every HTTP request.

→ Only available in Spring WebApplicationContext.

Ex: web controllers handling request-Specific data

```
@Scope("request")
@Component
public class MyRequestBean
    {
    }
```

(4) Session:

→ A single bean instance per HTTP Session.

→ useful for sorting user session data.

Ex:

```
@Scope("session")
@Component
public class MySessionBean
    {
    }
```

(5) application:

→ A Single bean instance per ServletContext (i.e., per web-application)

→ shared across all Sessions & requests.

Ex:

```
@Scope("application")
@Component
public class MyApplicationBean
{ }
```

(6) WebSocket:

→ A Single bean instance per WebSocket Session

→ useful for Web-Socket-based communication.

Ex:

```
@Scope("websocket")
@Component
public class MyWebSocketBean
{ }
```

* Introduction to AOP (Aspect oriented programming)

→ AOP (Aspect-oriented programming pattern that helps in modularizing cross-cutting concerns like logging, Security, transaction Management, profiling etc. which are usually Spread out in many different classes.

Why use AOP in Spring:

→ Keeps the business logic clean.

→ promotes code reusability

→ Easy to maintain update cross-cutting logic in one place.

Types of Advice in Spring AOP:

Advice Type	Description
→ @Before	Runs before the method execution
→ @After	Runs after the method completes (success (or) failure)
→ @AfterReturning	Runs after Successful execution
→ @AfterThrowing	Runs if method throws an exception

Example

Java

```
@Aspect
@Component
Public class LoggingAspect
    {
@Before ("execution(* com.example.service.*.*(..))")
    Public void logBefore (JoinPoint joinPoint)
        {
S.o.pln("calling method:" + joinPoint.getSignature().getName
        }
    }
```

→ The logs the method name before execution for any method in Com.example.service

* Implementing Aop Advices in Spring Boot:

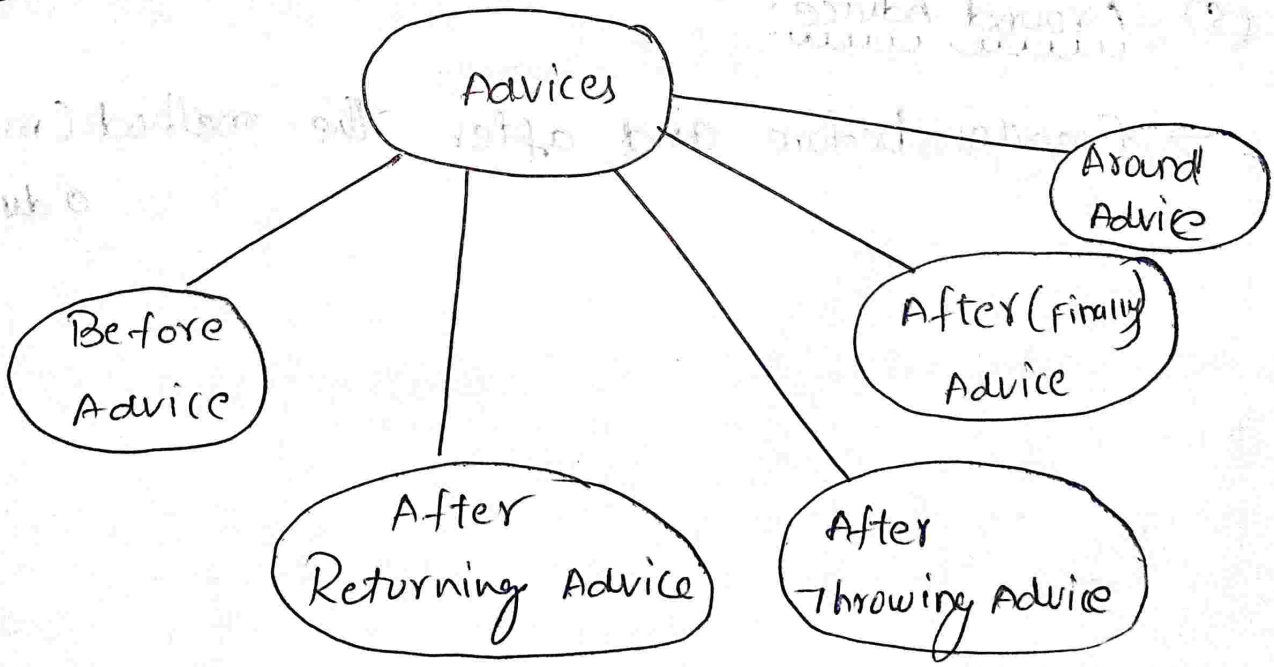
→ Aspect oriented programming (AOp) in Spring allows us to define cross-cutting concerns like

- logging
- Transactions
- Security

from bussiness logic

→ Advices are actions taken at Specific Join points in the application.

Types of Advices:



(1) Before Advice: -

→ Executes before the target method is invoked.

(2)- After Returning Advice:

→ Executes after the target method returns Successfully.

(3) After Throwing Advice:

→ Executes if the target method throws an exception.

(4) After (Finally) Advice:

→ Executes after method execution (whether Success (or) failure)

(5) Around Advice:

→ Executes before and after the method (most powerful advice).

Example Aspect with Advices:

```
@Aspect
@Component
public class LoggingAspect
{
@Before("execution(* com.example.demo.PaymentService.makePayment(..))") public void logBefore
(JoinPoint jointPoint)
{
S.o.ptn(" Before method:"+JoinPoint.getSignature().getName());
}

@AfterReturning("execution(* com.example.demo.PaymentService.makePayment(..))") public void logAfterReturning
(JoinPoint joinPoint)
{
S.o.ptn("AfterReturning from:"+joinPoint.getSignature().getName());
}
```

```
@AfterThrowing("execution(* com.example.demo.PaymentService.
                                    makePayment(..))")
        Public void logAfterThrowing(JoinPoint joinPo
        {
S.o.pln("Exception in method:" + joinPoint.getSignature().getName());
        }
@After("execution(*com.example.demo.PaymentService.makePayment(..))")
        public void logAfter(JoinPoint joinPoint)
        {
S.o.pln("After(Finally) method:" + joinPoint.getSignature().getName());
        }
@Around("execution(* com.example.demo.PaymentService.makePayment(..))")
    public object logAround(proceedingJoinpoint PJP) throws throwable
        {
S.o.pln("Around(Before) method:" + PJP.getSignature().getName());
        object result = PJP.proceed();
S.o.pln("Around (After) method:" + PJP.getSignature().
                                    getName());
        return result;
        } }
```

Output :

Around (Before) method : makePayment

Before method : makePayment

payment of 5000.0 made to account : 1234f

After Returning from : makePayment

After (Finally) method : makePayment

Around (After) method : makePayment

* Practices in Spring Boot Application:

Spring Boot Application:

→ Spring Boot is a framework built on top of the Java ~~Application~~ Spring Framework.

→ It helps in creating production-ready applications easily

↳ Less Configuration

↳ Auto configuration

↳ Embedded Server (Tomcat)

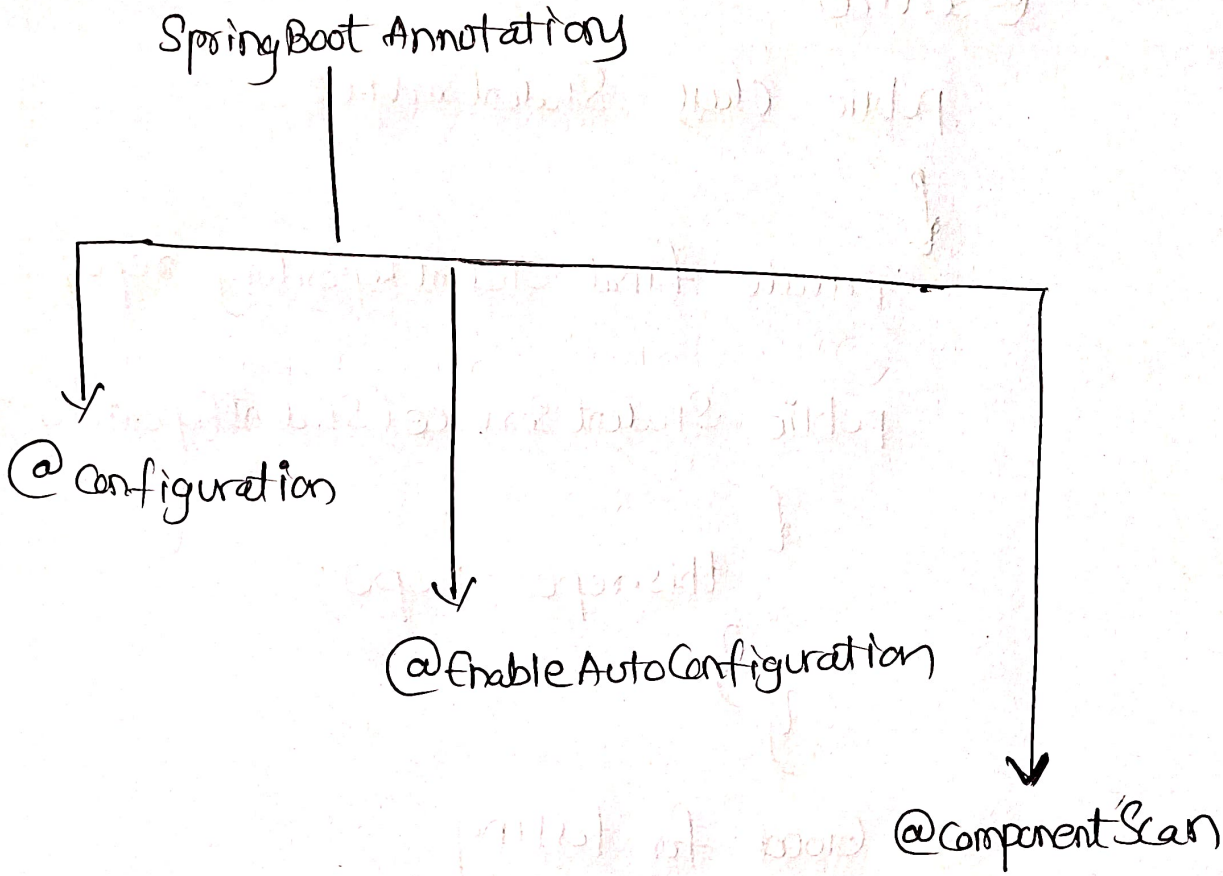
↳ Ready to run applications

Main class in Spring Boot:

→ The main entry point of a Spring Boot Application

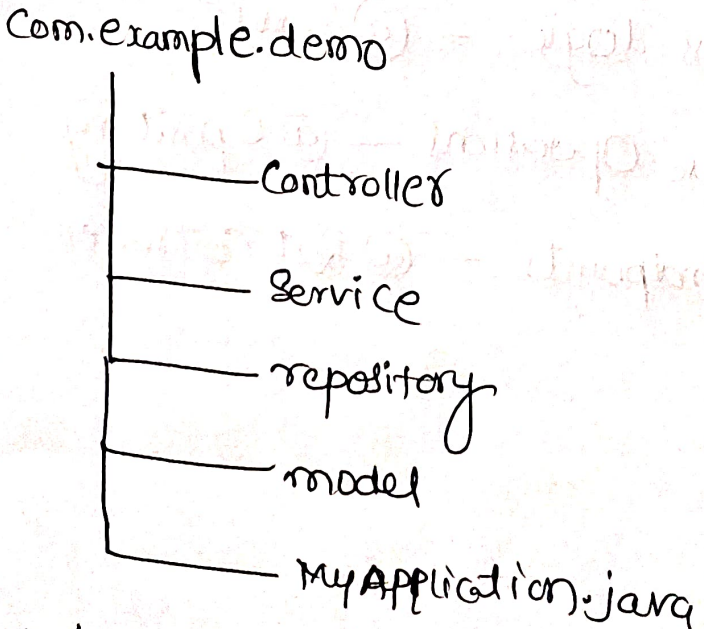
```
@SpringBootApplication
    public class MyApplication
      {
    public static void main (String arg[])
    {
SpringApplication.run(MyApplication.class, args);
    } }
```

@SpringBootApplication Combiny three annotations:



Important practices in SpringBoot Applications:

(1) Maintain a clean project structure



→ Layered architecture
→ improved readability.

2) Use Constructor Injection:

```
@Service
public class StudentService
{
    private final StudentRepository repo;

    public StudentService(StudentRepository repo)
    {
        this.repo = repo;
    }
}
```

→ Good for testing

→ objects remain immutable.

(3) Use @Service, @Repository, @Controller properly

→ Business logic – @Service

→ Database operations – @Repository

→ API endpoints – @RestController

* Autowiring in Spring Boot:

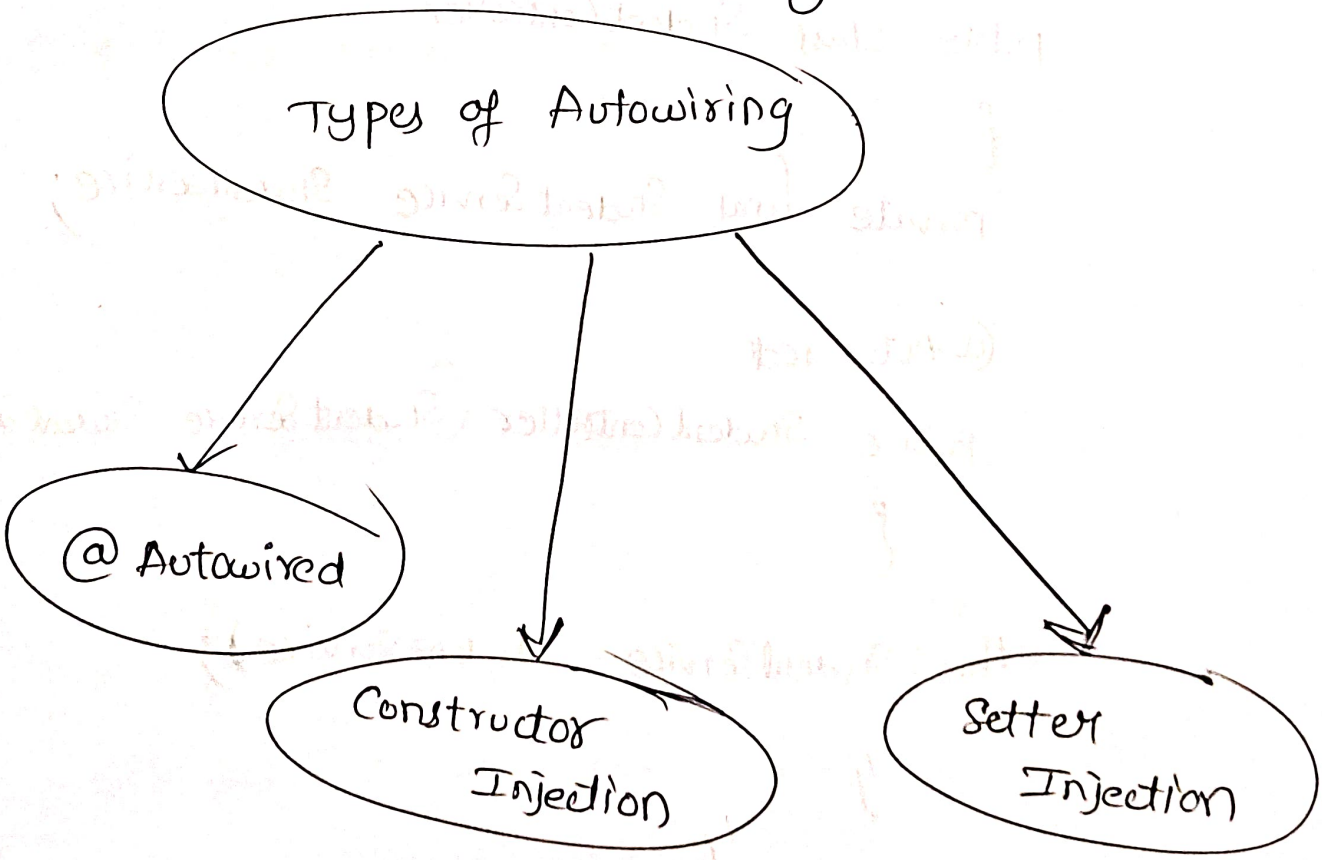
→ Autowiring means automatically injecting dependent beans into a class without explicitly creating objects using "new".

→ Spring Boot automatically detects and providing provides the required object by Searching in the Applicant Context.

(1) Helps avoid manual object creation.

(2) Reduces boilerplate code.

(3) Enable loose coupling.



(1) @Autowired:

→ Most commonly used.

→ Tells Spring to inject the required bean.

Example

```
@Autowired
private StudentService studentService;
```

(2) Constructor Injection:

→ Spring Boot recommends using constructor based autowiring.

```
@Service
public class StudentController
{
private final StudentService studentService;

@Autowired
public StudentController(StudentService studentService)
{
this.StudentService = studentService);
}
}
```

3) Setter Injection:

```
@Autowired
public void setStudentService(StudentService studentService)
{
    this.studentService = studentService;
}
```

Autowiring Best practices:

→ prefer Constructor injection

→ Avoid field injection in real projects.

→ Keep bean names clear & meaningful.