1. **Inversion of Control (IoC)**

* Prinzip in der Software Entwicklung, wo für die Manipulation von Objekten ein Container oder Framework zuständig ist
* Wird im OOP Kontext verwendet.
* Framework übernimmt die Kontrolle über Objekte.
* Wird noch „Hollywood Principle“ genannt: “You don’t call us, we call you.”

**Vorteile**:

* Implementierung wird von der Ausführung getrennt
* Leichter, zwischen Implementierungen zu wechseln
* Leichter zu testen, da Komponenten leichter zu isolieren sind

**IOC** wird durch Strategry Design pattern, Service Locator pattern, Factory pattern und Dependency Injection erreicht.

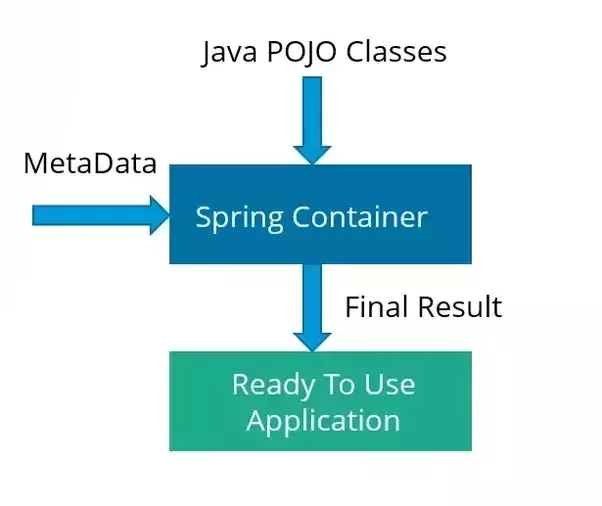
Ein **IOC Container** (**Dependency Injection Container**) ist ein Factory, welches die Dependency Injection erleichtert.

**Der Containter kümmert sich um:**

1. Erstellung von Objekten
2. Wiring von Objekten
3. Konfiguration von Objekten
4. Lifecycle von Objekten

**2 Containertypen in Spring:**

1. BeanFactory
   1. Liest Metadata aus dem XML
   2. Kleinere Anwendungen
2. ApplicationContext
   1. Fortgeschrittene Version von BeanFactory
   2. Größere Anwendungen



|  |
| --- |
| **BeanFactory:**  **Resource resource = new ClassPathResource("emp.xml");**  **BeanFactory factory = new XmlBeanFactory(resource);**  **Employee e1 = (Employee)container.getBean("em1");** |
| **ApplicationContext:**  **ApplicationContext container = new ClassPathXmlApplicationContext("emp.xml");**  **Employee e1 = (Employee)container.getBean("em1");** |
| **ConfigurableApplicationContext:**  **//implementation of the ApplicationContext**  **ConfigurableApplicationContext context = SpringApplication.run(SomeClassName.class, args);**  **context.getBean(QuickStart.class).run();** |

**Dependency Injection**  
Design Pattern zur Implementierung von IoC  
 Ein Objekt (Client) "erhält" ein anderes Objekt (Service), es instanziert es nicht selbst  
 Der Code, der dem Client den Service übergibt, ist der Injector  
 Der Client bestimmt nicht selbst, welchen Service er nutzt, das bestimmt der Injector  
   
kurzes Beispiel:  
   
public class Krankenpfleger{  
 private Kranker kranker = null;  
 public Krankenpfleger(Kranker k){  
 this.kranker = k;  
 }  
 }  
   
class Kranker{  
   
 private String name;  
 public Kranker(String name){  
 this.name = name;  
 }  
   
//...getters und setters  
 }  
   
Die Abhängigkeit wird über den Konstruktor injeziert!  
 Sehr ungünstig wäre der Folgende Konstruktor:  
   
public class Krankenpfleger{  
   
 private Kranker kranker = null;  
 public Krankenpfleger(){  
 this.kranker = new Kranker("Peter");  
 }  
 }  
   
Hier wird die Abhängigkeit nicht von außen über einen Injektor gesteuert, sondern direkt von   
 der Klasse Krankenpfleger. -> strong coupling -> very bad  
Dependendy Injection ist also im Grunde genommen eine Parameterübergabe.  
   
Vorteile:  
 Client ist gut konfigurierbar  
 besser fürs UnitTesting  
 weniger kopplung

**Beans**

* Objekte, die vom Spring IoC Container gemanagt (initialisiert, konfiguriert und zusammengestellt) werden
* Beans fungieren in Java unter anderem als Container zur Datenübertragung, weshalb sie als Vorlage für alle verwalteten Ressourcen dienen
* Beans in Spring können entweder über XML oder mit Annotations in Java konfiguriert werden
* Bei der Annotation-Konfiguration können Beans über die Annotations @Component, @Service, @Controller oder @Repository oberhalb der Klasse definiert werden
* Konfiguration von Beans -> eclipse