**Spring Framework**

**Spring is a lightweight, open-source application development framework** that uses Java Enterprise Edition (Java EE) and Enterprise JavaBeans model (EJB).

Spring is often thought of as a “**framework of frameworks**” because it supports other integrated frameworks in the Java ecosystem like Struts, Hibernate, and Tapestry. Spring comes in many variants that are tailored to particular uses such as Spring MVC, Spring Boot, or Spring Security.

**Spring is known for its dependency injection and Inversion of Control (IoC) systems, which allow us to create large-scale, loosely-coupled applications with ease**. Spring is especially suited for financial and enterprise applications due to its speed, security, and easy-to-build transaction systems.

Overall, **Spring Framework** **allows us to create enterprise-scale applications** **that are secure and can automatically handle low-level functionalities** like maintaining container lifecycles and managing dependencies.

### **Advantages of Spring**

* **Plain Old Java Object (POJO)**: **Spring applications just use normal Java objects** rather than servers or enterprise containers. This **makes applications more lightweight**.
* **Modularity IoC and** [**MVC**](https://www.educative.io/blog/mvc-tutorial) allow us to **reuse components across the application** **without manually managing the dependencies of each.**
* **Flexible Configs**: Can **opt to use Java-based annotations or XML configuration.**
* **Secure**: **Easy to implement security modules with** [**authentication**](https://www.educative.io/blog/kerberos-in-5-minutes) **and validation features**.
* **Scalable transactions**: Spring offers a **consistent, scalable transaction management interface for both local transactions and global transactions**. This includes caching protocols.
* **Strong support for Java ecosystem**: **Spring makes innovative uses of pre-existing technologies**, such as ORM frameworks, JEE, and JDK timers

### **Disadvantages of Spring**

* **Complexity**: Spring Framework **opts for increased control and complexity over ease of use**. Spring is, therefore, **harder to pick up than other web frameworks**. While very powerful, Spring is best suited for experienced developers that can handle its steep learning curve.
* **Parallel Mechanisms**: Spring’s **wide range of options** means a **single task can be accomplished in several ways**. We’ll need intricate knowledge of our available tools to choose the ideal solution and avoid cross-team confusion.
* **Lacking robust documentation**: Spring’s **documentation is missing clear guidelines on a number of topics**, especially around cybersecurity techniques. It’s up to users to deal with common attacks like cross-site forgery and scripting attacks.

## **Features of Spring**

* **Inversion of Control (IoC)**: **Shifts responsibility for dependencies from objects to the framework itself**. Without IoC, each object must handle its own instance of dependency. **With IoC**, **the object simply lists that it needs a certain dependency and the framework configures and manages it** **automatically**. This simplifies the app development process by consolidating the dependency system and reducing the workload on individual classes.
* **Aspect-Oriented Programming**: **Spring** **AOP** **allows us to have different class loaders to increase modularity and separate cross-cutting concerns**.
* **Dependency Injection**: **The process Spring uses to automatically manage the dependencies of beans**. Spring checks which beans are required for a particular bean to function and injects them as a dependency. Spring can **perform dependency injection by using** **a** **constructor** or **a** **setter method**.
* **Container Lifecycle Management**: **Automatically creates, manages, and configures application objects**.
* **MVC support**: Supports the MVC web architecture that separates functionalities between model, view, and controller layers.
* **Error Handling**: Includes JDBC exception handling with an exception hierarchy system.

## **Spring Architecture and Components**

Let’s break down the fundamental pieces of a Spring Framework application to help see the big picture.

### **Beans**

**Spring Beans** are **instances of classes that are managed by Spring**. They are the most fundamental component of a Spring program.

Traditionally, objects would create personal instances of their dependencies. **Spring manages all the dependencies of an object and instantiates the object into a bean after injecting the required dependencies**. The @Component annotation is the most common method of defining beans.



**Beans have a set lifecycle and visibility based on their scope**. There are **six types of scopes**: **singleton**, **prototype**, **request**, **session**, **application**, and **websocket**. The **singleton and prototype scopes can be used in any application** while the **last four scopes are only available for a web application**.

The **default scope of a bean is singleton**, in which **only one instance of the bean is created and cached in memory**. **Multiple requests for the bean return a shared reference to the same bean**. In contrast, **prototype scope results in the creation of new beans** whenever a request for the bean is made to the application context.

### **Autowiring**

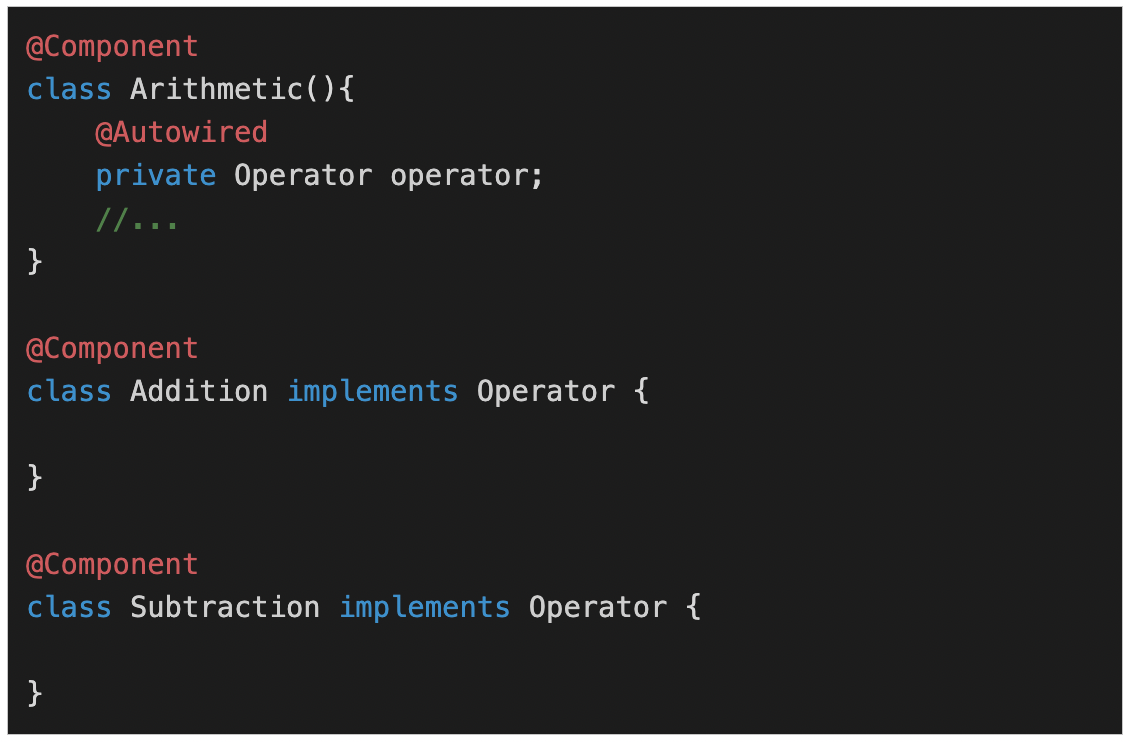
Spring’s **method of injecting beans** into one and another. Spring **identifies a selected bean’s dependency**, **looks for a match**, and **then populates the dependency**. The @Autowired annotation tells Spring to **find and inject a collaborating bean into another**.

**@Autowired is applied to**

* **field**: for the field-based dependency injection
* **setter** for the setter dependency injection. Same as field-based dependency injection.
* **constructor** for constructor-based dependency injection

If **more than one bean of the same type is available**, **Spring throws an error**.

In the following scenario, two beans of type Operator are detected by Spring:



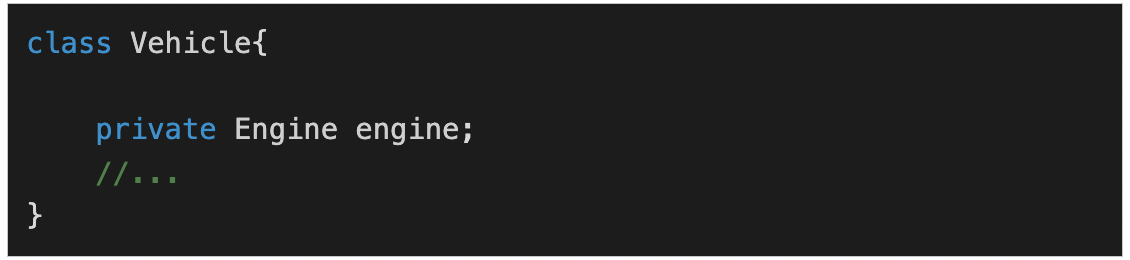
**Difference between constructor based and setter based DI**

* **Injection of dependencies can be optional or mandatory**. For **mandatory injection**, **we use constructor-based DI**. While for the **optional dependencies we can use setter-based DI**. However, we **can mark a setter-based DI with @Required** annotation.
* In the **case of cyclic dependency**, **constructor-based DI won’t be able to inject** but **setter-based DI would be able to inject**.
* If **more parameters are to be injected** then it is **advisable to use constructor-based DI**.

### **IoC Container**

An **IoC container is a framework** that provides the Inversion of Control functionality. The **IoC container manages the beans and creates any instances required**.

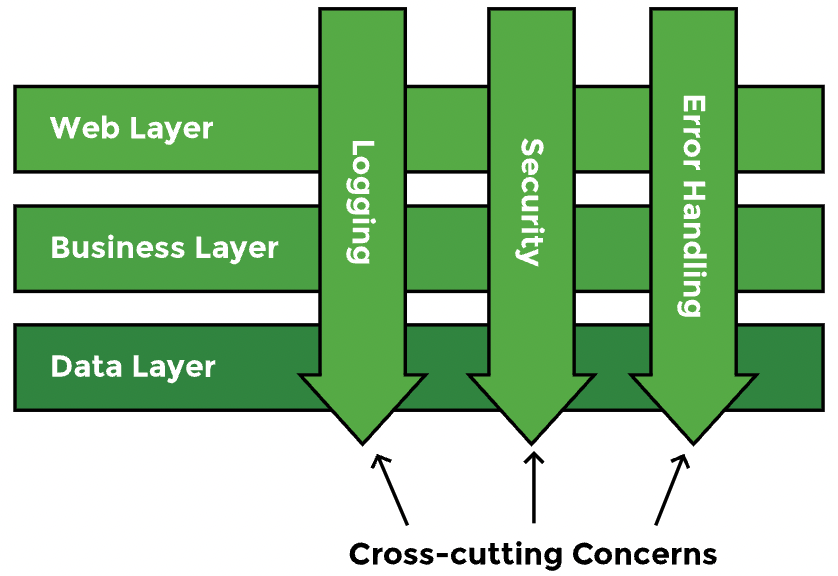
For example, say we have the class Vehicle that depends on another class, Engine.



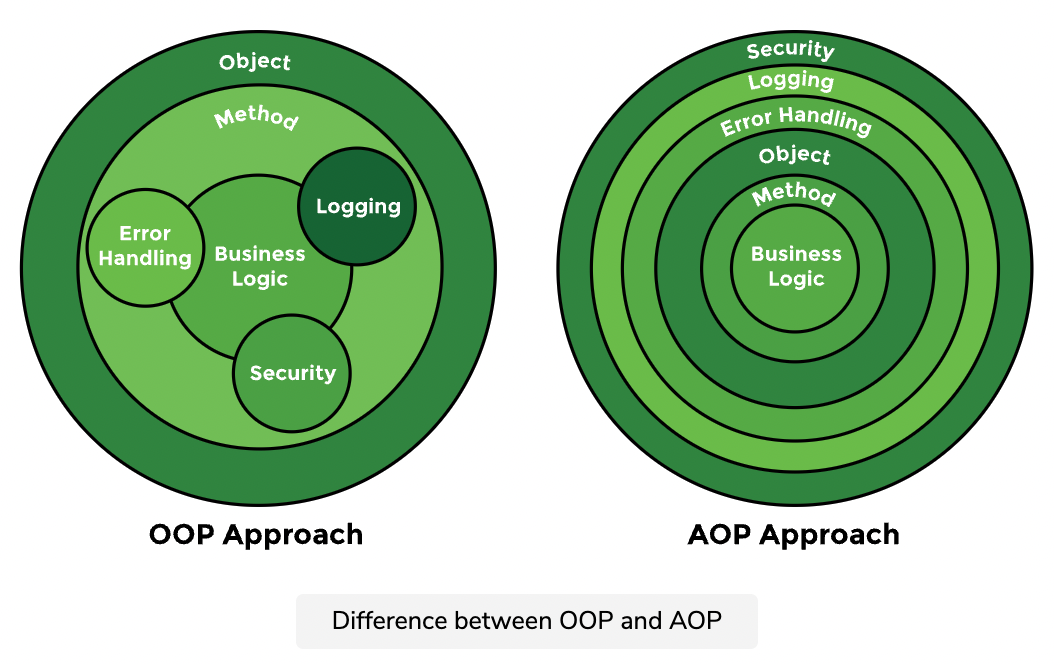
With IoC, the **programmer only needs to mention the dependency** and Spring will do the rest. The **IoC container will create instances of both classes** and **inject** Engine **into** Vehicle **as a** **dependency**.

### **Spring Aspect-Oriented Programming**

**Aspect-Oriented Programming (AOP)** is the **best approach for implementing cross-cutting concerns**. Applications are divided into layers like web, business, data, etc. Each layer works independently. There are **some concerns that are common across layers**. These include **security, logging, transaction management, auditing, error handling, performance tracking**, etc. These concerns are present in all the layers and are thus called cross-cutting concerns.



**AOP provides an easy way to add concerns like printing logs or tracking the performance of methods across layers**. For example, we may want to print logs of methods that provide a certain feature. These methods may belong to the web, business, or data layer. **Using the OOP approach, we can call the logger from these methods**. The problem with such an approach is that **in the future, if the logs are not needed anymore**, or if they are needed for another group of methods, **we will have to make changes to the source code**. By following the **AOP approach**, **such changes are easy to maintain**. **Instead of changing the source code, concerns are implemented separately**. Thus, **any cross-cutting concern can be added and removed without recompiling the complete code by only changing the config files**. The logging functionality can be defined separately and applied to any method belonging to any layer easily.



## 

## **Spring Validations**

Spring Boot’s Bean Validation support comes with the validation starter, which we can include in our project:



However, if we have also included the web starter, the validation starter comes with it:



Basically, **Bean Validation** **works by defining constraints to the fields of a class by annotating them with certain** [**annotations**](https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/javax/validation/constraints/package-summary.html).

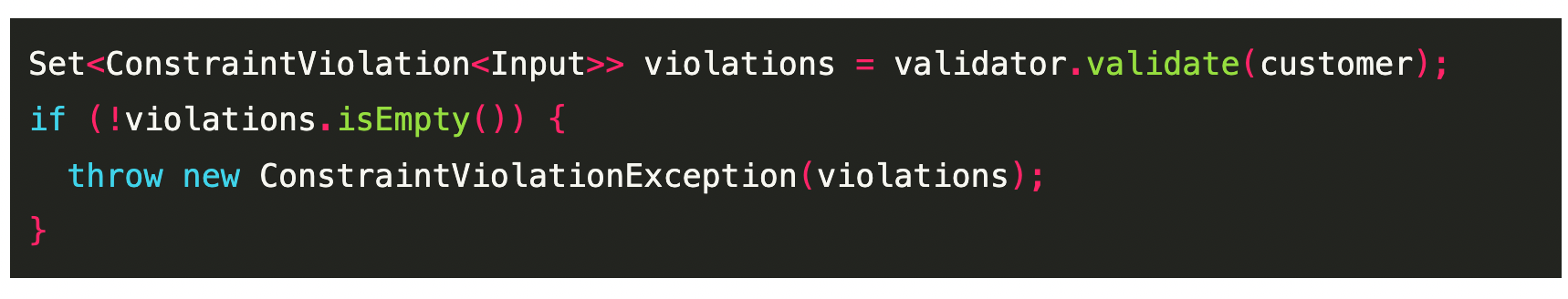
Some of the most common validation annotations are:

* **@NotNull**: to say that a **field must not be null**.
* **@NotEmpty**: to say that a **list field must not be empty**.
* **@NotBlank**: to say that a **string field must not be an empty** string (i.e. it must have at least one character).
* **@Min and @Max**: to say that a **numerical field is only valid when its value is above or below a certain value**.
* **@Pattern**: to say that a **string field is only valid when it matches a certain regular expression**.
* **@Email**: to say that a **string field must be a valid email address**.

An example of such a class would look like this:



**To validate if an object is valid**, we pass it into a [**Validator**](https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/javax/validation/Validator.html) which **checks if the constraints are satisfied**:



### 

### **@Validated and @Valid**

In many cases, **Spring does the validation for us**. We don’t even need to create a validator object ourselves. Instead, **we can let Spring know that we want to have a certain object validated**. This works **by using the @Validated and @Valid annotations**.

The **@Validated** annotation is a **class-level annotation** **that we can use to tell Spring to validate parameters that are passed into a method of the annotated class**.

We can put the **@Valid** annotation on **method parameters and fields** **to tell Spring that we** **want a method parameter or field to be validated**.

