# Introduction to Spring

Spring is an open-source, Java-based, Inversion of Control (IoC) container and application framework that was originally released in 2002. Spring was originally created as a competitor to J2EE (Java 2 platform, Enterprise Edition), it has grown massively in the last 16 years with the addition of over 20 additional “projects” covering such areas as: accessing data in a consistent manner (Spring Data), deploying microservice applications to the cloud (Spring Cloud), securing applications (Spring Security) and creating Spring applications with next to no configuration that “just run” (Spring Boot).

# Inversion of Control

Inversion of control (**IoC**) is a programming principle that “inverts” the “flow of control” by handing control of certain aspects of the program (usually Objects or blocks of code) to a framework (in this case Spring) where the framework then becomes responsible for managing these aspects and will create, initialise, destroy, etc. them all by itself.

Any object that an IoC container is responsible for managing is referred to as a **bean.**

## Dependency Injection

Dependency injection (**DI**) is one *implementation* of Inversion of Control. With this method the container passes managed objects (**beans**) into other objects by matching the name of the dependency to the name of a bean.

Using dependency injection increases modularity and reduces coupling because it allows you to have a very generic dependency and swap out the actual implementation at will.

There are three main methods of Dependency Injection:

### Field Injection

* Dependencies are injected into fields marked with the **@Autowired** annotation.
* Generally considered bad practice because classes have to be modified at run time (**reflection**) as this method does not use a setter or a constructor.

### Setter Injection

* Dependencies are injected using **setter** methods.
* Much better practice than **Field Injection** but has a problem in that it allows for circular dependencies.
* Because dependencies are injected **after** object creation you can’t be certain that a created object will have received all of its dependencies.

### Constructor Injection

* Generally considered to be best practice due to the efficiency of not using **reflection**, not allowing for **circular dependencies** and for ensuring the existence of dependencies in created objects.
* Dependencies are injected via object constructors so they are present when the object is created.
* Constructors can get quite bloated using this method but this is typically a sign that the class has too many responsibilities.

### Example Injection

Follow this link to find a simple example of the three forms of **Dependency Injection** done via Java configuration:

<https://gist.github.com/JHarry444/ce95e10be1c158e1128cc2bb848915a6>

## Dependency Lookup

Dependency lookup differs from dependency injection as it involves the object searching for its own dependency by name rather than having the framework dynamically determine which bean should be “injected”.

This does have the advantage of ensuring the specificity of the dependency but is more resource intensive as a result. Typically, Dependency Injection is used rather than Dependency Lookup as it is more flexible and less resource-intensive.

## Wiring

Wiring is the process by which Spring matches up the Bean dependency with the bean it is going to be injected. The **@Autowired** annotation tells Spring that it should match the beans automatically without the user having to specify where the bean is coming from explicitly; it does this by matching either the type or the name of the dependency to a bean that is present in the container. Autowiring can be risky as you aren’t guaranteed to get the right bean injected and if there is more than one possible bean available in the container then Spring may well just throw an error.

# Bean Scopes

“The scope of a bean defines the life cycle and visibility of that bean in the contexts in which it is used.”

In short, a bean’s scope denotes how long it is kept in the container, some beans will only be used once before being recreated whereas others may be used across the application.

## Singleton

There is only going to be one instance of this bean created during the entire runtime of the application and any references to this bean will point at the same object.

## Prototype

Creates a new instance of the bean every time the bean is referenced.

## Request

Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.

## Session

Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

## Application

Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically, only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.