The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework’s IoC container. The [BeanFactory](https://docs.spring.io/spring-framework/docs/6.2.3/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object. [ApplicationContext](https://docs.spring.io/spring-framework/docs/6.2.3/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds:

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, persistence layer objects such as repositories or data access objects (DAOs), presentation objects such as Web controllers, infrastructure objects such as a JPA EntityManagerFactory, JMS queues, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of repositories and business logic to create and load domain objects.

The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. By using the method T getBean(String name, Class<T> requiredType), you can retrieve instances of your beans.

The ApplicationContext lets you read bean definitions and access them, as the following example shows:

You can then use getBean to retrieve instances of your beans. The ApplicationContext interface has a few other methods for retrieving beans, but, ideally, your application code should never use them. Indeed, your application code should have no calls to the getBean() method at all and thus have no dependency on Spring APIs at all. For example, Spring’s integration with web frameworks provides dependency injection for various web framework components such as controllers and JSF-managed beans, letting you declare a dependency on a specific bean through metadata (such as an autowiring annotation).

**Bean Overview**

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML <bean/> definitions).

Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:

* A package-qualified class name: typically, the actual implementation class of the bean being defined.
* Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
* References to other beans that are needed for the bean to do its work. These references are also called collaborators or dependencies.
* Other configuration settings to set in the newly created object — for example, the size limit of the pool or the number of connections to use in a bean that manages a connection pool.

This metadata translates to a set of properties that make up each bean definition. The following table describes these properties:

Java Configuration

If you use Java Configuration, a corresponding @Bean method always silently overrides a scanned bean class with the same component name as long as the return type of the @Bean method matches that bean class. This simply means that the container will call the @Bean factory method in favor of any pre-declared constructor on the bean class