

A large, light gray play button icon is positioned on the left side of the slide. It consists of a white right-pointing triangle centered within a series of concentric circles in shades of gray.

Beans and Dependency Injection

1. Components and beans
2. A closer look at components and beans
3. Dependency injection
4. A closer look at dependency injection

1. Components and Beans

- Overview of components
- Defining a component
- Component scanning in Spring Boot
- Accessing a bean

Overview of Components

- In Spring, a **component** is:
 - A class that Spring will automatically instantiate
- To define a component in Spring, annotate a class with any of the following annotations:
 - `@Component`
 - `@Service`
 - `@Repository`
 - `@Controller/@RestController`

Defining a Component

- Here's an example of how to define a component:

```
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class MyComponent {
```

```
    ...
```

```
}
```

`MyComponent.java`

- Spring will automatically create an instance of this class
 - The instance is known as a "bean"

Component Scanning in Spring Boot

- When a Spring Boot app starts, it scans for component classes
 - It looks in the application class package, plus sub-packages
- You can tell Spring Boot to look elsewhere if necessary:

```
@SpringBootApplication( scanBasePackages={"mypackage1", "mypackage2"} )  
public class Application {  
    ...  
}
```

Accessing a Bean

- When a Spring Boot application starts up, it creates beans and stores them in the "application context"
- You can access beans in the application context as follows:

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

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        MyComponent bean = ctx.getBean(MyComponent.class);
        System.out.println(bean);
    }
}
```

`Application.java`

2. A Closer Look at Components and Beans

- Specifying a name for a component
- Understanding singleton scope
- Getting a singleton-scope bean
- Lazily instantiating a singleton bean
- Defining a different scope
- Getting prototype-scope beans

Specifying a Name for a Component

- Every bean has a name
 - By default, it's the name of the component class (with the first letter in lowercase)
- You can specify a different name for the bean as follows
 - When Spring creates a bean, it will be named `myCoolBean`

```
import org.springframework.stereotype.Component;  
  
@Component("myCoolBean")  
public class SomeComponent {  
    ...  
}
```


Understanding Singleton Scope

- By default, Spring creates a single bean instance
 - i.e., the default scope is "singleton"
- You can annotate with `@Scope("singleton")` if you want to be explicit:

```
@Component  
public class MySingletonComponent { ... }
```



· Equivalent

```
@Component  
@Scope("singleton")  
public class MySingletonComponent { ... }
```

Getting a Singleton-Scope Bean

- Singleton beans are created at application start-up
 - For each call to `getBean()`, you get the same bean

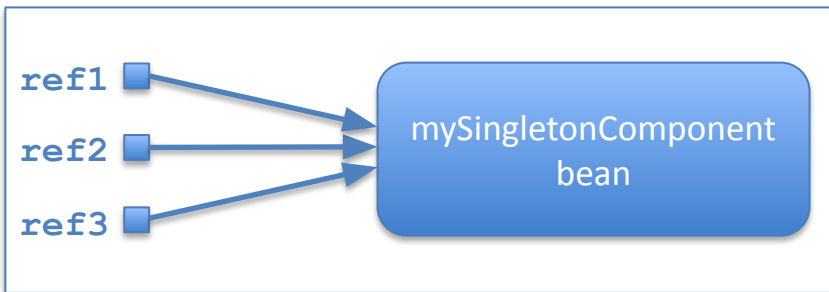
```
ApplicationContext ctx = SpringApplication.run(Application.class, args);
```

```
MySingletonComponent ref1 = ctx.getBean(MySingletonComponent.class);
```

```
MySingletonComponent ref2 = ctx.getBean(MySingletonComponent.class);
```

```
MySingletonComponent ref3 = ctx.getBean(MySingletonComponent.class);
```

`Application.java`



Lazily Instantiating a Singleton Bean

- You can tell Spring to lazily instantiate a singleton bean
 - Annotate the component class with `@Lazy`

```
@Component
@Lazy
public class MySingletonComponent {
    ...
}
```

- Avoids creating beans until needed
 - Speeds start-up time

Defining a Different Scope

- You can use `@Scope` to specify the scope for a bean:

```
@Component  
@Scope("prototype")  
public class MyPrototypeComponent { ... }
```

`MyPrototypeComponent.java`

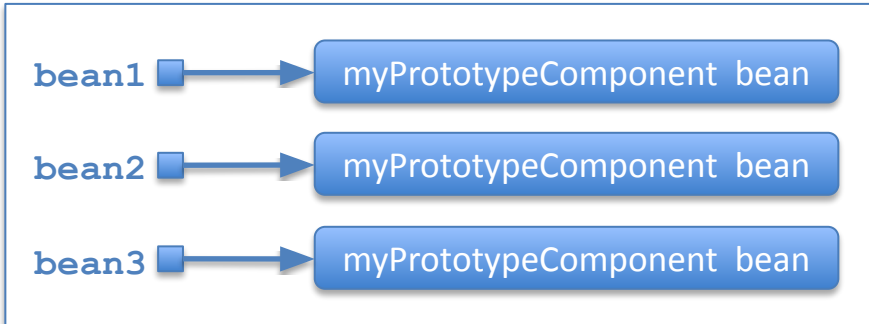
- There are several scopes available:
 - "prototype"
 - "request"
 - "session"
 - "application"

Getting Prototype-Scope Beans

- Consider this example of getting prototype beans
 - For each call to `getBean()`, Spring creates a new bean

```
ApplicationContext ctx = SpringApplication.run(Application.class, args);  
  
MyPrototypeComponent bean1 = ctx.getBean(MyPrototypeComponent.class);  
MyPrototypeComponent bean2 = ctx.getBean(MyPrototypeComponent.class);  
MyPrototypeComponent bean3 = ctx.getBean(MyPrototypeComponent.class);
```

`Application.java`

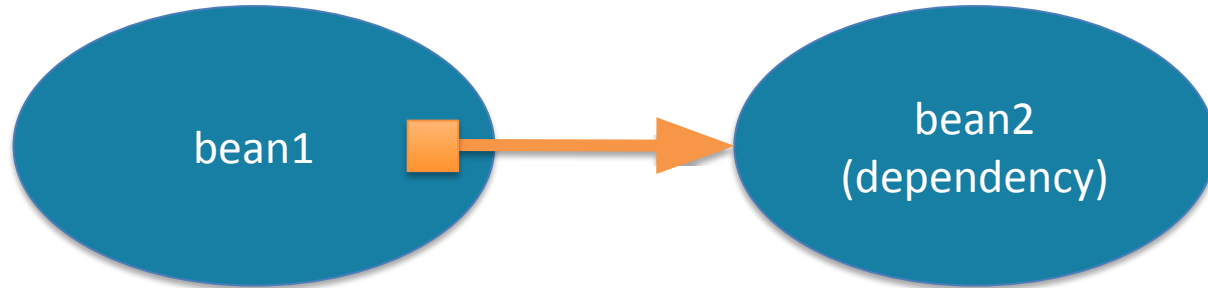


3. Dependency Injection

- Overview of dependency injection
- Injecting dependencies into fields
- Injecting dependencies into a constructor
- Fine-tuning autowiring

Overview of Dependency Injection

- Dependency Injection (DI) is a key Spring concept
 - Use configuration to describe dependencies between components
- Spring automatically injects dependencies into beans
 - This is known as "autowiring"



Injecting Dependencies into Fields

- If a bean has dependencies...
 - You can inject via `@Autowired`
- You can use `@Autowired` on a field
 - Spring injects a bean of the specified type into the field

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private BankRepository repository;

    ...
}
```

BankServiceImpl.java

Injecting Dependencies into a Constructor

- You can also use `@Autowired` on a constructor
 - Spring will inject beans into all constructor parameters

```
@Service
public class BankServiceImpl implements BankService {

    private BankRepository repository;

    @Autowired
    public BankServiceImpl(BankRepository repository) {
        this.repository = repository;
    }
    ...
}
```

BankServiceImpl.java

- Note: If a component only has one constructor, you can omit `@Autowired` (Spring autowires ctor params automatically)

Fine-Tuning Autowiring

- You can specify which bean instance to inject
 - Use `@Qualifier` to specify the bean name you want

```
@Autowired
@Qualifier("primaryRepository")
private BankRepository repository;
```

- You can mark an `@Autowired` member as optional
 - Set `required=false`

```
@Autowired(required=false)
private BankRepository repository;
```

4. A Closer Look at Dependency Injection

- Autowiring a collection
- Autowiring a map
- Injecting values into beans
- Specifying values in application properties
- Aside: Common application properties

Autowiring a Collection

- You can autowire a `Collection<T>`
 - Spring injects a collection of all the beans of type `T`
- Example
 - Autowire a collection of all beans that implement the `BankRepository` interface

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private Collection<BankRepository> repositories;

    ...
}
```

Autowiring a Map

- You can also autowire a `Map<String, T>`
 - Spring injects a map indicating all beans of type `T`
 - Keys are bean names, values are bean instances
- Example
 - Autowire `BankRepository` names/beans

```
@Service
public class BankServiceImpl implements BankService {

    @Autowired
    private Map<String, BankRepository> repositoriesMap;

    ...
}
```

Injecting Values into Beans

- You can inject values into beans, via @Value
 - Use `$` to inject an application property value
 - Use `#` to inject a general value via Spring Expression Language

```
import org.springframework.beans.factory.annotation.Value;
...

@Component
public class MyBeanWithValues {

    @Value("${name}")           // Inject value of "name" application property.
    private String name;

    @Value("#{ 5 * 7.5 }")      // Inject general Java value via SpEL.
    private double workingWeek;
    ...
}
```

MyBeanWithValues.java

Specifying Values in Application Properties

- You can define values in the application properties file

```
name=John Smith
```

```
application.properties
```

- Here's how to access the bean in the main code

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        ...
        MyBeanWithValues beanWithValues = ctx.getBean(MyBeanWithValues.class);
        System.out.println(beanWithValues);
    }
}
```

```
Application.java
```

Aside: Common Application Properties

- Spring Boot defines lots of common application properties by default - you can see the full list here:
 - <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>
- You can override any of these properties in your code
 - In `application.properties` or `application.yml`

A large, light gray play button icon is positioned on the left side of the slide. It consists of a white right-pointing triangle centered within a series of concentric circles in shades of gray.

Summary

- Components and beans
- A closer look at components and beans
- Dependency injection
- A closer look at dependency injection

Exercise



- Define another component class that implements the `BankRepository` interface
 - Name the class `BankRepositoryCheckedImpl`
 - The class rejects transactions greater than a threshold amount
 - Specify the threshold in `application.properties`
- Modify `BankServiceImpl` to make use of this new component class
- Write some code in `main()` to test the new functionality