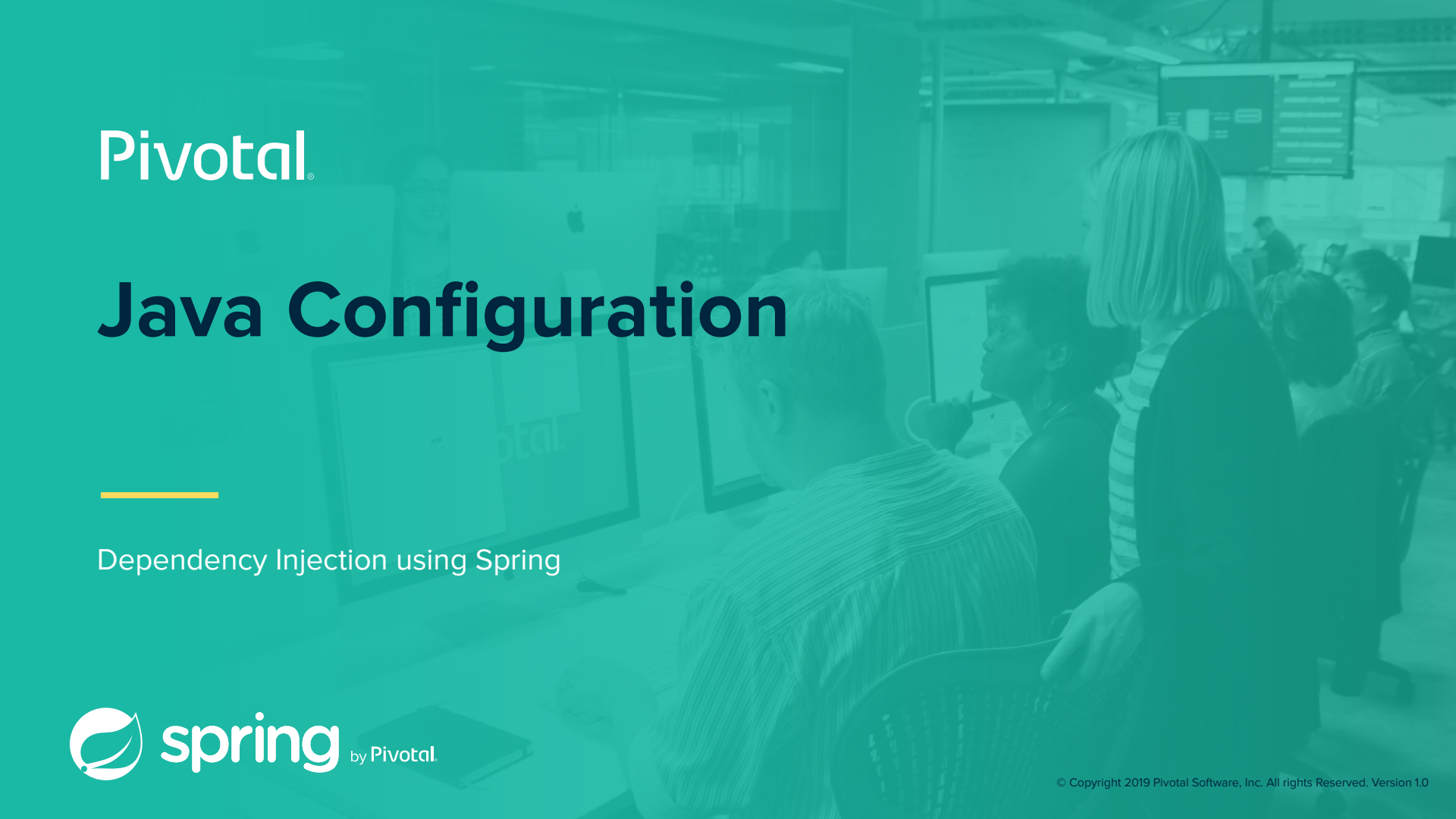


Pivotal

Java Configuration

Dependency Injection using Spring

spring by Pivotal

Module Objectives

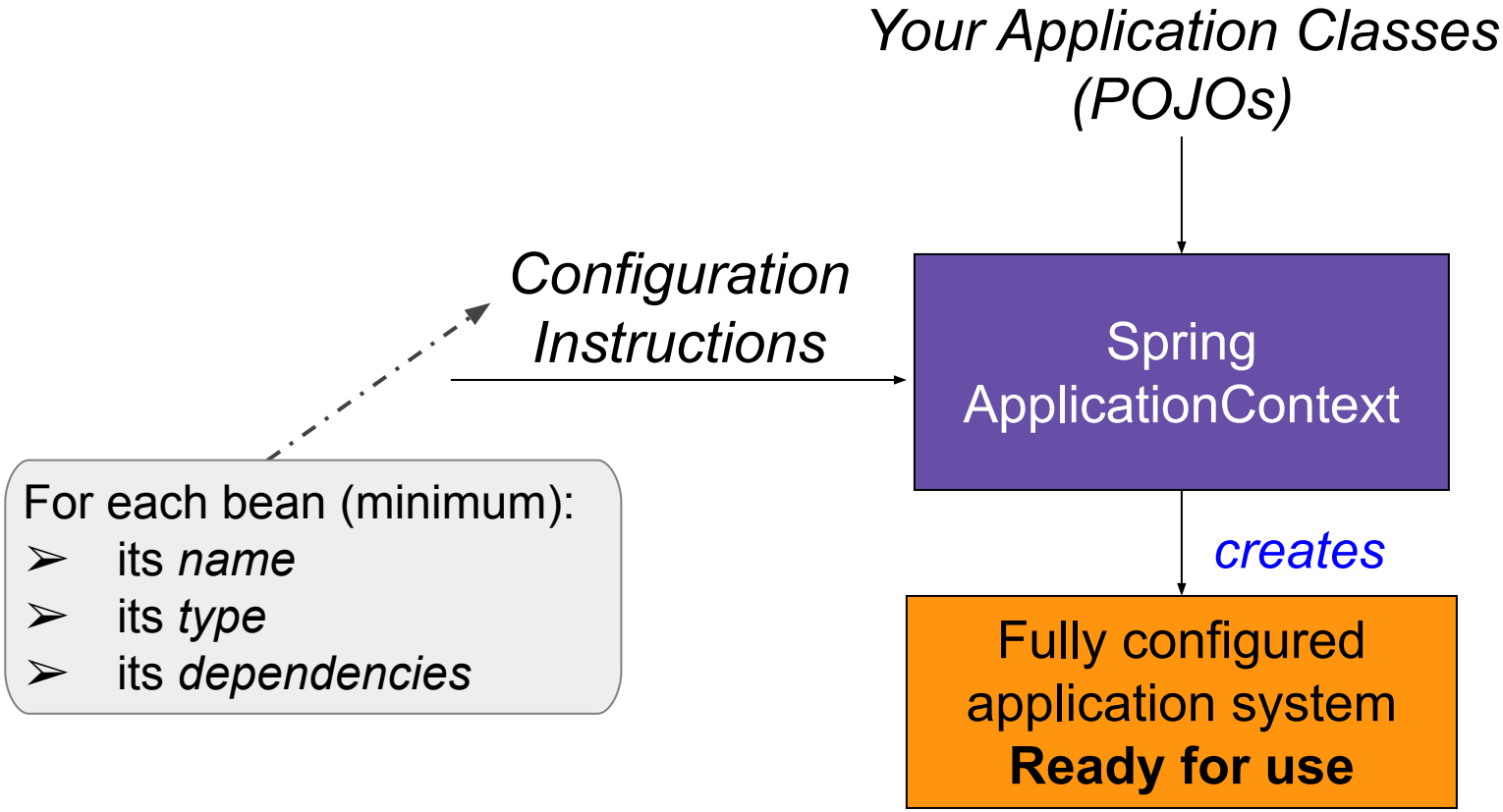
After completing this lesson, you should be able to do the following

- Define Spring Beans using Java code
- Access Beans in the Application Context
- Handle multiple Configuration files
- Handle Dependencies between Beans
- Explain and define Bean Scopes

Agenda

- **Spring quick start**
- Creating an application context
- Multiple Configuration Files
- Bean scope

How Spring Container Works



Your Application Classes as POJO's with Dependencies

```
public class TransferServiceImpl implements TransferService {
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    …
}
```

Dependency: Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements AccountRepository {
    public JdbcAccountRepository(DataSource ds) {
        this.dataSource = ds;
    }
    …
}
```

Dependency: Needed to access account data in the database

You do not have to use *interfaces* to define Spring Beans, but it is a good Java practice as they encourage loose-coupling.

Configuration Instructions with @Configuration & @Bean

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService() {
    return new TransferServiceImpl( accountRepository() );
  }
  @Bean public AccountRepository accountRepository()  {
    return new JdbcAccountRepository( dataSource() );
  }
  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
    dataSource.setUsername("transfer-app");
    dataSource.setPassword("secret45");
    return dataSource;
  }
}
```

Creating and Using the Application

```
// Create application context from the configuration
ApplicationContext context =
    SpringApplication.run( ApplicationConfig.class );

// Look up a service
TransferService service =
        context.getBean("transferService", TransferService.class);

// Use the service
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

What configuration to use to define beans

Bean ID
Based in method name

Note that Spring will create *four* beans: **`ApplicationConfig`** is *also* a Spring Bean - it is used to create the others.

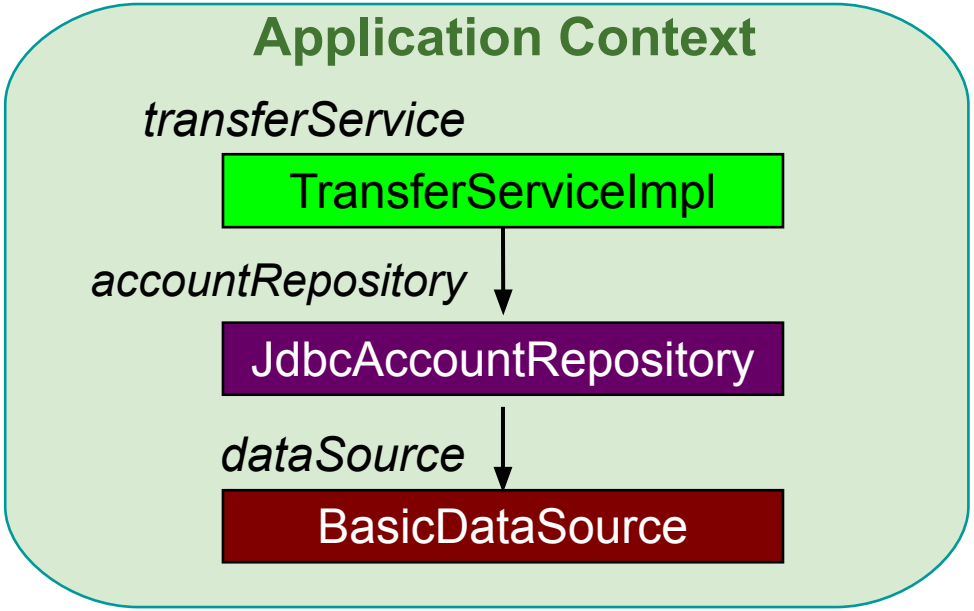
Accessing a Bean Programmatically

Multiple options

```
ApplicationContext context = SpringApplication.run(...);

// Use bean id, a cast is needed
TransferService ts1 = (TransferService)  context.getBean("transferService");

// Use typed method to avoid casting
TransferService ts2 =  context.getBean("transferService", TransferService.class);

// No need for bean id if type is unique
TransferService ts3 =  context.getBean(TransferService.class );
```

Inside the Spring Application Context

```
// Create application context from the configuration
ApplicationContext context = SpringApplication.run( ApplicationConfig.class )
```



Quick Start Summary

- Spring separates application configuration from application objects (beans)
- Spring manages your application objects
  - Creating them in the correct dependency order
  - Ensuring they are fully initialized before use
- Each bean is given a unique id / name
  - Should reflect service or role the bean provides to clients
  - Bean ids should not contain implementation details

Agenda

- Spring quick start
- **Creating an application context**
- Multiple Configuration Files
- Bean scope

Creating a Spring Application Context

- Spring application context can be bootstrapped in any environment, including
  - JUnit system test
  - Web application
  - Standalone application

Application Context Example

Instantiating Within an Integration or System Test

```
public class TransferServiceTests {
    private TransferService service;

    @BeforeEach public void setUp() {
        // Create application context from the configuration
        ApplicationContext context =
            SpringApplication.run( ApplicationConfig.class )
        // Look up a service
        service = context.getBean(TransferService.class);
    }

    @Test public void moneyTransfer() {
        Confirmation receipt =
            service.transfer(new MonetaryAmount("300.00"), "1", "2"));
        Assert.assertEquals("500.00", receipt.getNewBalance());
    }
```

Bootstraps the system to test

Tests the system

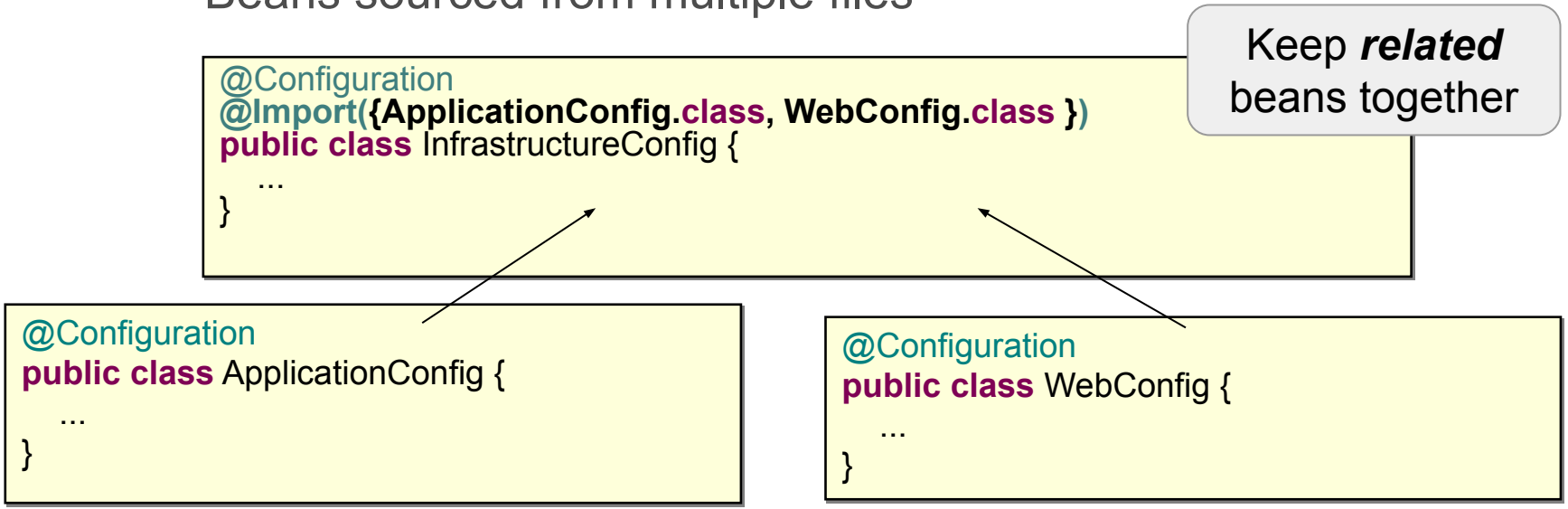
Using JUnit 5 – JUnit 4 or TestNG also supported

Agenda

- Spring quick start
- Creating an application context
- **Multiple Configuration Files**
- Bean scope

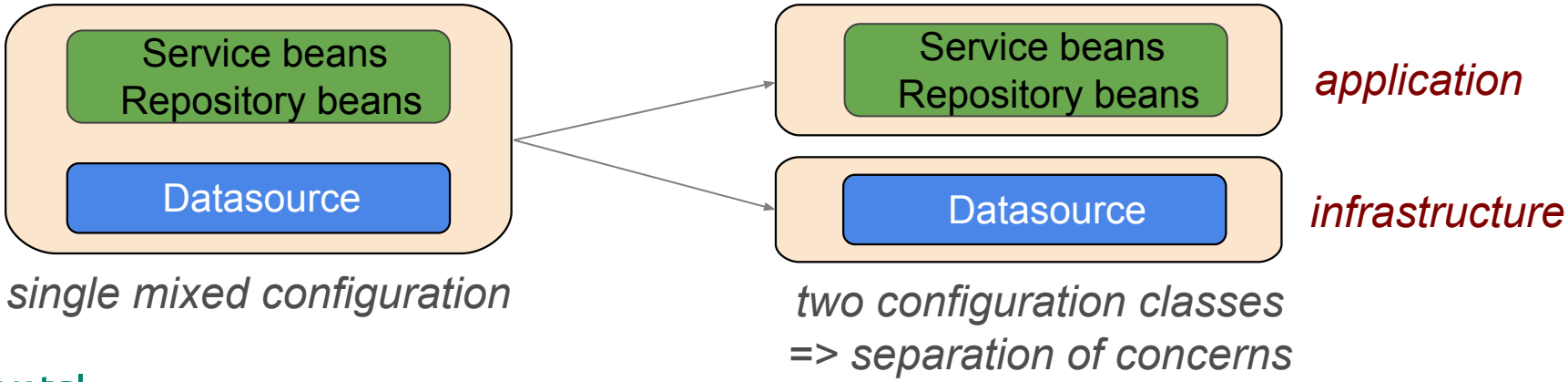
Creating an Application Context from Multiple Configurations

- Your **`@Configuration`** class can get too big
  - Instead use *multiple* config. files combined with **`@Import`**
  - Defines a single Application Context
    - Beans sourced from multiple files



Creating an Application Context from Multiple Files

- *Separation of Concerns* principle
  - Keep related beans in the same @Configuration
- *Best Practice:* separate “application” & “infrastructure”
  - Infrastructure often changes between environments



Mixed Configuration

```
@Configuration
public class ApplicationConfig {

    @Bean public TransferService transferService()
        { return new TransferServiceImpl( accountRepository() ); }

    @Bean public AccountRepository accountRepository()
        { return new JdbcAccountRepository( dataSource() );  }

    @Bean public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("org.postgresql.Driver");
        dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
        dataSource.setUsername("transfer-app");
        dataSource.setPassword(“secret45” );
        return dataSource;
    }
}
```

application beans

Coupled to a local Postgres environment

infrastructure bean

Partitioning Configuration

application beans

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService(AccountRepository repo) {
    return new TransferServiceImpl ( repo );
  }

  @Bean public AccountRepository accountRepository(DataSource ds) {
    return new JdbcAccountRepository( ds );
  }
}
```

```
@Configuration
@Import(ApplicationConfig.class )
public class TestInfrastructureConfig {
  @Bean public DataSource dataSource() {
    ...
  }
}
```

Infrastructure config imports all the others

infrastructure bean

```
ApplicationContext ctx = SpringApplication.run( TestInfrastructureConfig.class )
```

Referencing Dependencies 1 - Via Autowired

- Use @Autowired to inject a bean defined elsewhere

```
@Configuration
public class ApplicationConfig {
  private final DataSource dataSource;

  @Autowired
  public ApplicationConfig(DataSource ds) {
    this.dataSource = ds;
  }

  @Bean
  public AccountRepository accountRepository() {
    return new JdbcAccountRepository( dataSource );
  }
}
```

```
@Configuration
@Import(ApplicationConfig.class )
public class InfrastructureConfig {
  @Bean
  public DataSource dataSource() {
    DataSource ds = new BasicDataSource();
    ...
    return ds;
  }
}
```

Referencing Dependencies 2 - Via Arguments

- *Alternative:* Define @Bean method arguments
  - Spring finds bean that matches type & injects the argument

```
@Configuration
public class ApplicationConfig {
  @Bean
  public AccountRepository accountRepository( DataSource dataSource ) {
    return new JdbcAccountRepository( dataSource );
  }
}
```

```
@Configuration
@Import(ApplicationConfig.class )
public class InfrastructureConfig {
  @Bean  public DataSource dataSource() {
    DataSource ds = new BasicDataSource();
    ...
    return ds;
  }
}
```

… But Avoid “Tramp Data”

Bad: dataSource is a “*tramp*”!

```
@Configuration
public class ApplicationConfig {
  @Bean public AccountService accountService( DataSource ds ) {
    return new AccountService( accountRepository(ds) );
  }

  @Bean public AccountRepository accountRepository( DataSource ds ) {
    return new JdbcAccountRepository( ds );
  }
}
```

tramp

tramp

Better: Pass *actual* dependency

```
@Configuration
public class ApplicationConfig {
  @Bean  public AccountService accountService( AccountRepository repo ) {
    return new AccountService( repo );
  }
  @Bean public AccountRepository accountRepository( DataSource ds ) {
    return new JdbcAccountRepository( ds );
  }
}
```

Bean Overriding

Recall: bean name from method name

- Defines same bean more than once
  - Allows override when testing: you get the *last* bean Spring sees defined
- Disabled by default from Spring Boot 2.1
  - To prevent a bean being accidentally overridden
  - Set *spring.main.allow-bean-definition-overriding=true* to enable it

```
@Configuration
public class Config1 {
  @Bean
  public String example() {
    return new String("example1");
  }
}
```

```
@Configuration
public class Config2 {
  @Bean
  public String example() {
    return new String("example2");
  }
}
```

```
@Import({ Config1.class, Config2.class })
public class TestApp {
  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(TestApp.class);
    System.out.println("Id=" + context.getBean("example"));
  }
}
```

Console output is *Id=example2*

Agenda

- Spring quick start
- Creating an application context
- Multiple Configuration Files
- **Bean scope**

Bean Scope: Default

service1 == service2

- Default scope is *singleton*

```
@Bean
public AccountService accountService() {
    return ...
}
```

These are equivalent

```
@Bean
@Scope("singleton")
public AccountService accountService() {
    return ...
}
```

One single instance

```
AccountService service1 = (AccountService)  context.getBean("accountService");
AccountService service2 =  (AccountService)  context.getBean("accountService");
assert service1 == service2;  // True – same object
```

Implications for Singleton Beans

- Typical Spring application – back-end web-server
  - Multiple requests in parallel
    - Handled by multiple threads
  - **Implications:**
    - Multiple threads accessing singleton beans *at the same time*

- Consider multi-threading issues
  - Stateless or Immutable beans
  - `synchronized` (harder)
  - Use a different scope

Bean Scope: prototype

service1 != service2

- Scope "*prototype*"
  - New instance created every time bean is referenced

```
@Bean
@Scope("prototype")
 public Action deviceAction() {
     return …
}
```

@Scope(scopeName="prototype")

```
Action action1 = (Action)  context.getBean("deviceAction");
Action action2 =  (Action)  context.getBean("deviceAction");
assert action1 != action2;  // True – different objects
```

TWO instances

Common Spring Scopes

- The most commonly used scopes are:

singleton	A single instance is used
prototype	A new instance is created each time the bean is referenced
session	A new instance is created once per user session - web environment only
request	A new instance is created once per request – web environment only

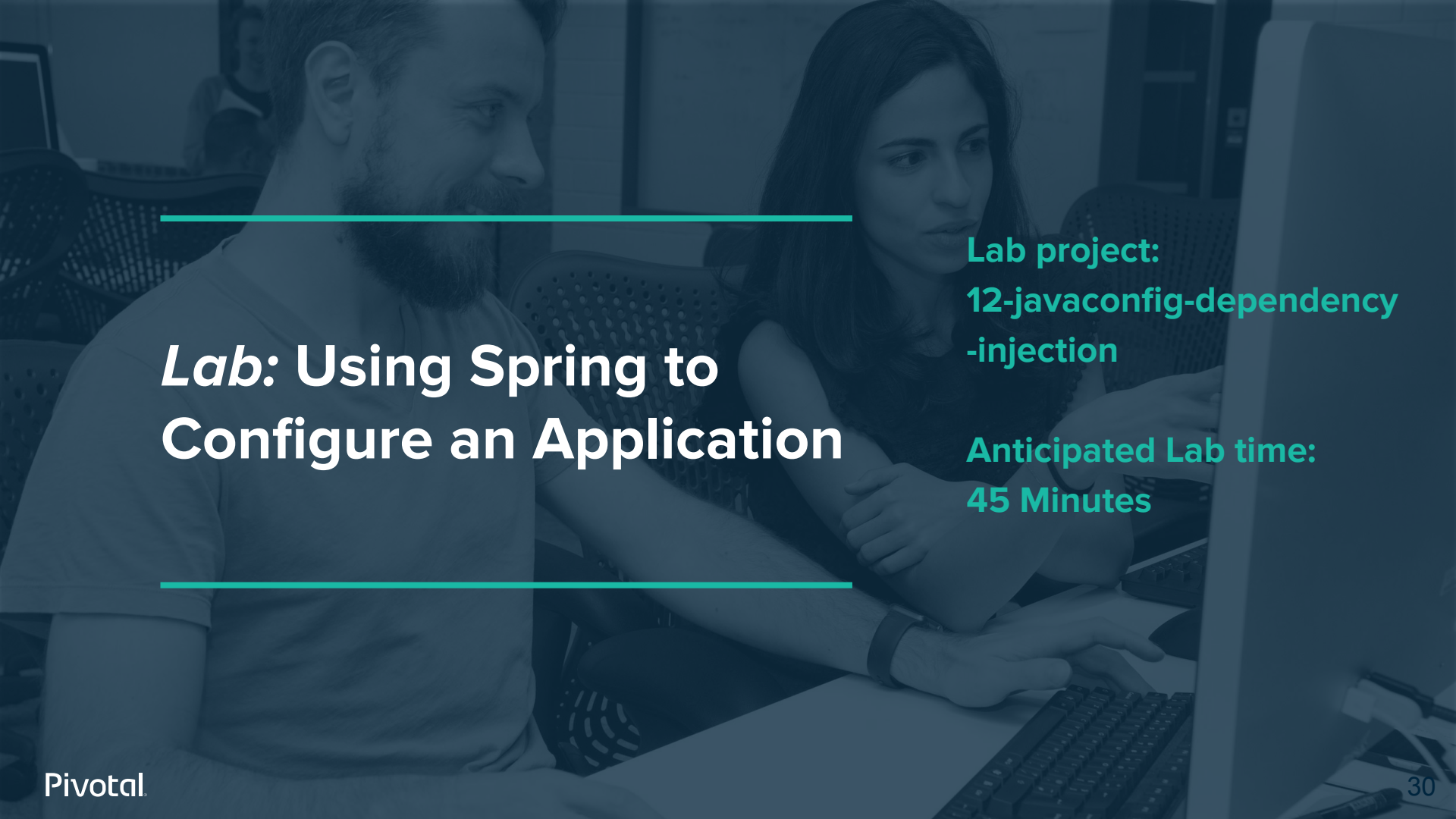
Other Scopes

- Spring has other more specialized scopes
  - Web Socket scope
  - Refresh Scope
  - Thread Scope (defined but not registered by default)
- Custom scopes (rarely)
  - You define a factory for creating bean instances
  - Register to define a custom scope name

These scopes are not covered by this course, but see Scope reference slide at end of this section

Dependency Injection Summary

- Your object is handed what it needs to work
  - Frees it from the burden of resolving its dependencies
  - Simplifies your code, improves code reusability
- Promotes programming to interfaces
  - Conceals implementation details of dependencies
- Improves testability
  - Dependencies easily stubbed out for unit testing
- Allows for centralized control over object lifecycle
  - Opens the door for new possibilities



Lab: Using Spring to Configure an Application

Lab project:
12-javaconfig-dependency-injection

Anticipated Lab time:
45 Minutes

Reference: Available Scopes

Scope	Description
singleton	Lasts as long as its **`ApplicationContext`**
prototype	**`getBean()`** returns a new bean every time. Lasts as long as you refer to it, then garbage collected
session	Lasts as long as user's HTTP session
request	Lasts as long as user's HTTP request
application	Lasts as long as the ServletContext (Spring 4.0)
global	Lasts as long as a global HttpSession in a *Portlet* application (*obsolete from Spring 5*)
thread	Lasts as long as its thread – defined in Spring but *not registered* by default
websocket	Lasts as long as its websocket (Spring 4.2)
refresh	Can outlive reload of its application context. Difficult to do well, assumes Spring Cloud *Configuration Server*