Programming in Java

6. Spring Framework

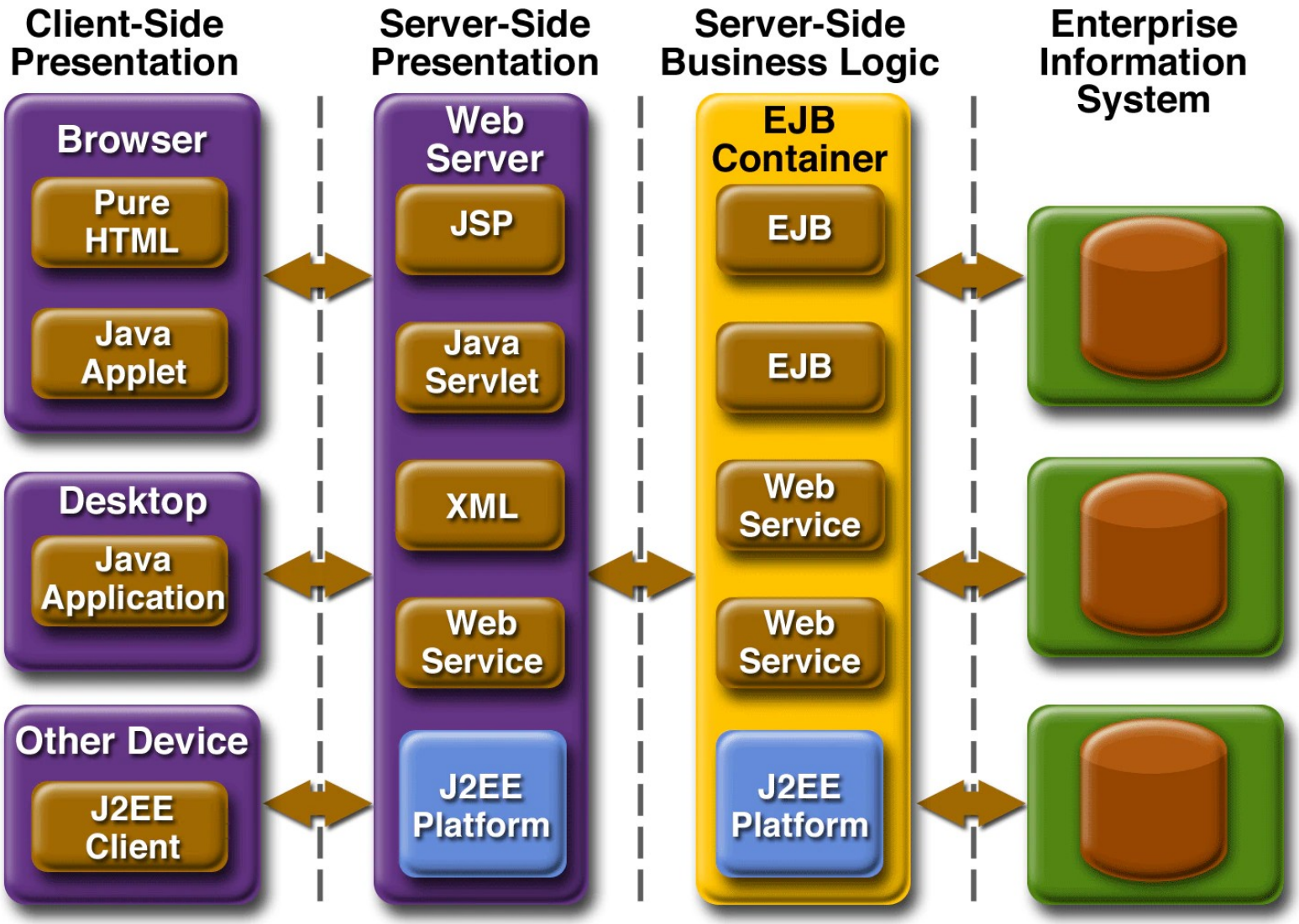
Application Architecture

- So far, we have been being building very simplistic Java applications
- As applications get more complex it becomes increasingly difficult to:
  - Manage the creation and organization of objects.
  - Ensure that all the object dependencies are satisfied
  - Deploy and run complex applications
  - Satisfy the non-functional requirements.
  - Load, stress, fail-over, throughput, etc.
- This is no Java solution because this is a software architecture problem, not a programming problem.

J2EE

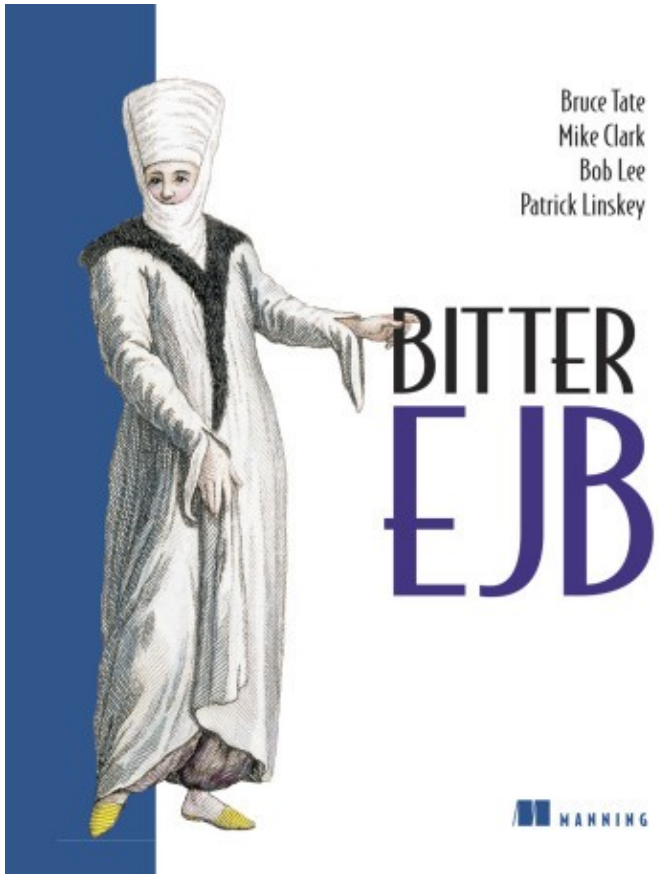
- SUN tried to solve the problem by introducing an “enterprise architecture” called J2EE
  - It was based on application servers like Tomcat into which runtime containers were deployed
  - The containers contained a deployable form of a Java application called an Enterprise Java Bean (EJB)
  - These were wrappers around one of more POJOs to make them deployable onto an application server
  - Very popular due to the Web components like Java Server Pages and Servlets
- But… EJBs were a failure
  - The required a lot of configuration and boilerplate code
  - And they performed horribly
  - They were too heavy-weight and complex for the value they provided

J2EE

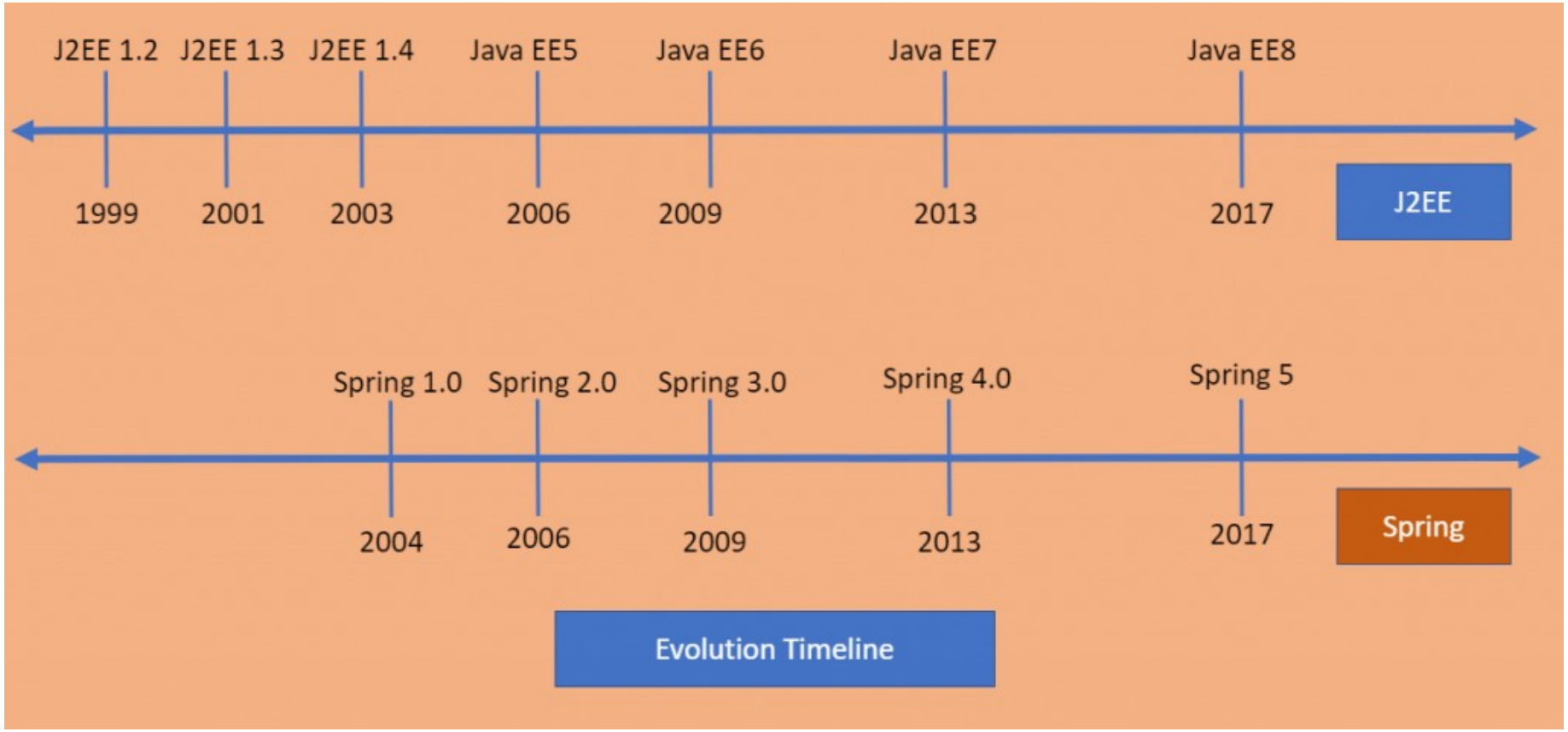

Client-Side Presentation
Server-Side Presentation
Server-Side Business Logic
Enterprise Information System
Browser
Pure HTML
Java Applet
Desktop
Java Application
Other Device
J2EE Client
Web Server
JSP
Java Servlet
XML
Web Service
J2EE Platform
EJB Container
EJB
EJB
Web Service
Web Service
J2EE Platform

The Spring Insurgency

- S2EE was so problematic to use that Rod Johnson created the first version of Spring as an alternative
  - The first version was in 2004 when frustration with J2EE was at its peak
  - Some of the developers of EJBs wrote a book describing how to work around the design problems of EJBs
- Spring became the go-to alternative to J2EE



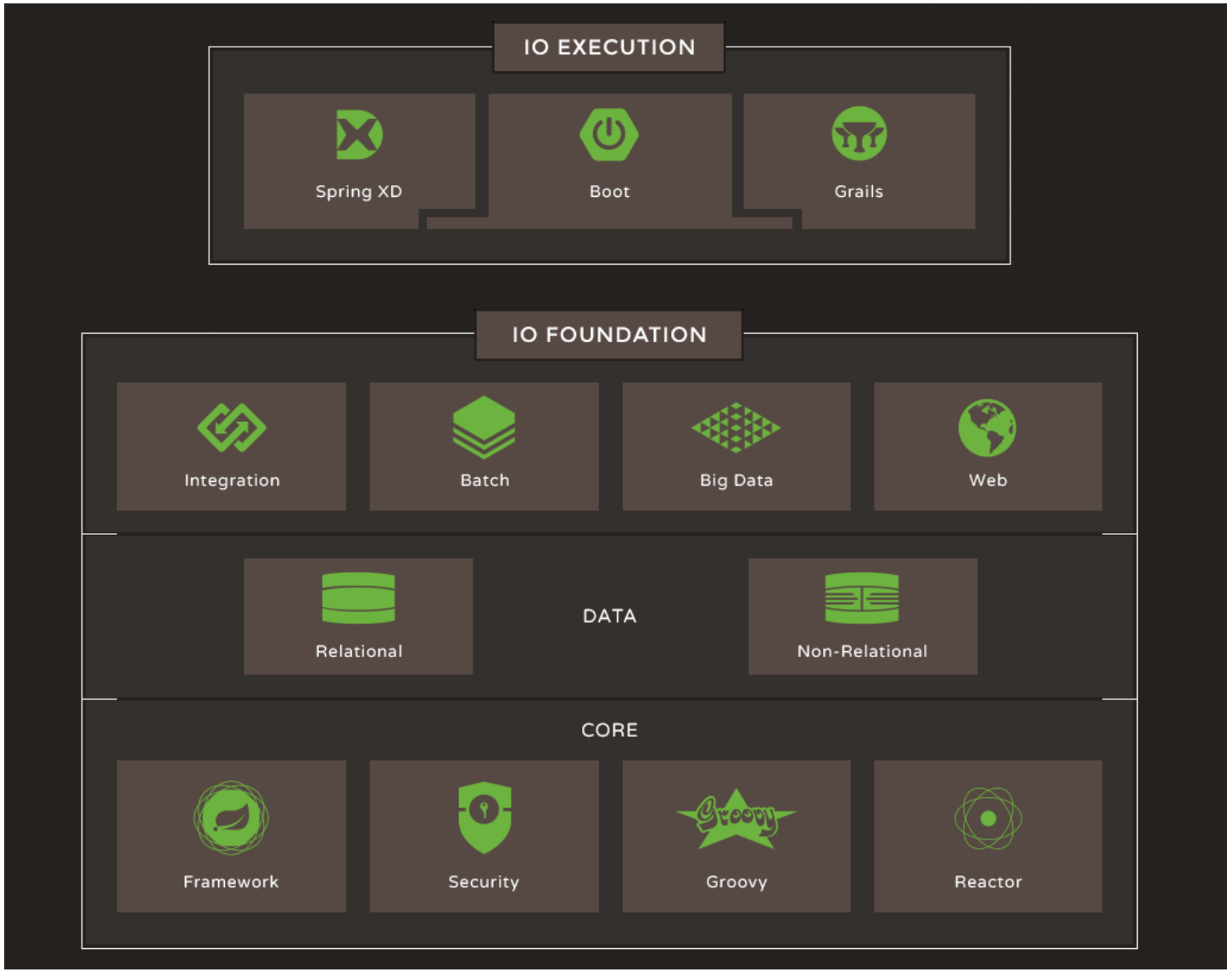
The Spring Timeline



The Spring Impact

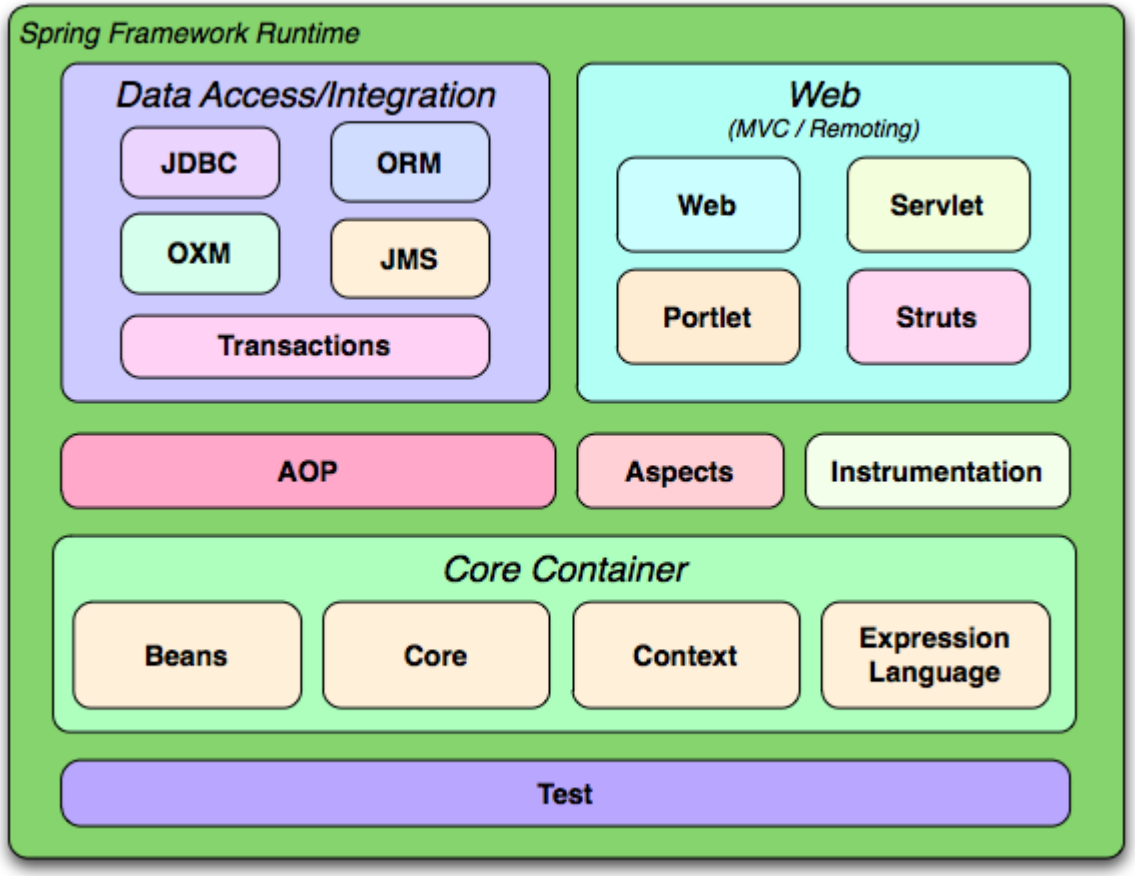
- After J2EE was rebranded as Java EE
  - Many of the features of Spring were adopted by Java EE
  - Especially the techniques of Inversion of Control (IoC) and Dependency Injection (DI)
- Spring has evolved into an ecosystem
  - Supports a number of projects that cover a wide range of development needs
  - Similar to the Apache project ecosystem
- This collection of projects is referred to as the Spring Platform
  - More info at http://spring.io

The Spring Platform



The Spring Framework

- Organized into a series of modules
  - Each provides a specific sort functionality
  - In the second week you saw the JPA data module at work
- The core container is the heart of the Spring Framework
  - It manages all the different services
  - Wires them together to form a larger application
  - Manages the low level operations like Java object creating and lifecycles



Inversion of Control

- Objects define their dependencies only through
  - Arguments to the object constructor
  - Arguments to a factory method
  - Properties set on an instance by a factory method
- Implementing the dependencies is done via Dependency Injection
  - Commonly done in constructors or setter methods
  - Dependencies are “injected” during or after the instance is created
- IoC means that the instance controls the creation of its dependencies or connections to existing instances it depends on
  - Dilemma, how does an object create the dependencies it needs before it’s created?
  - How does an object reuse an existing object it’s dependent on?
  - We solve the problem by delegating that sort of “wiring” of object together to a container that has the responsibility of resolving those references

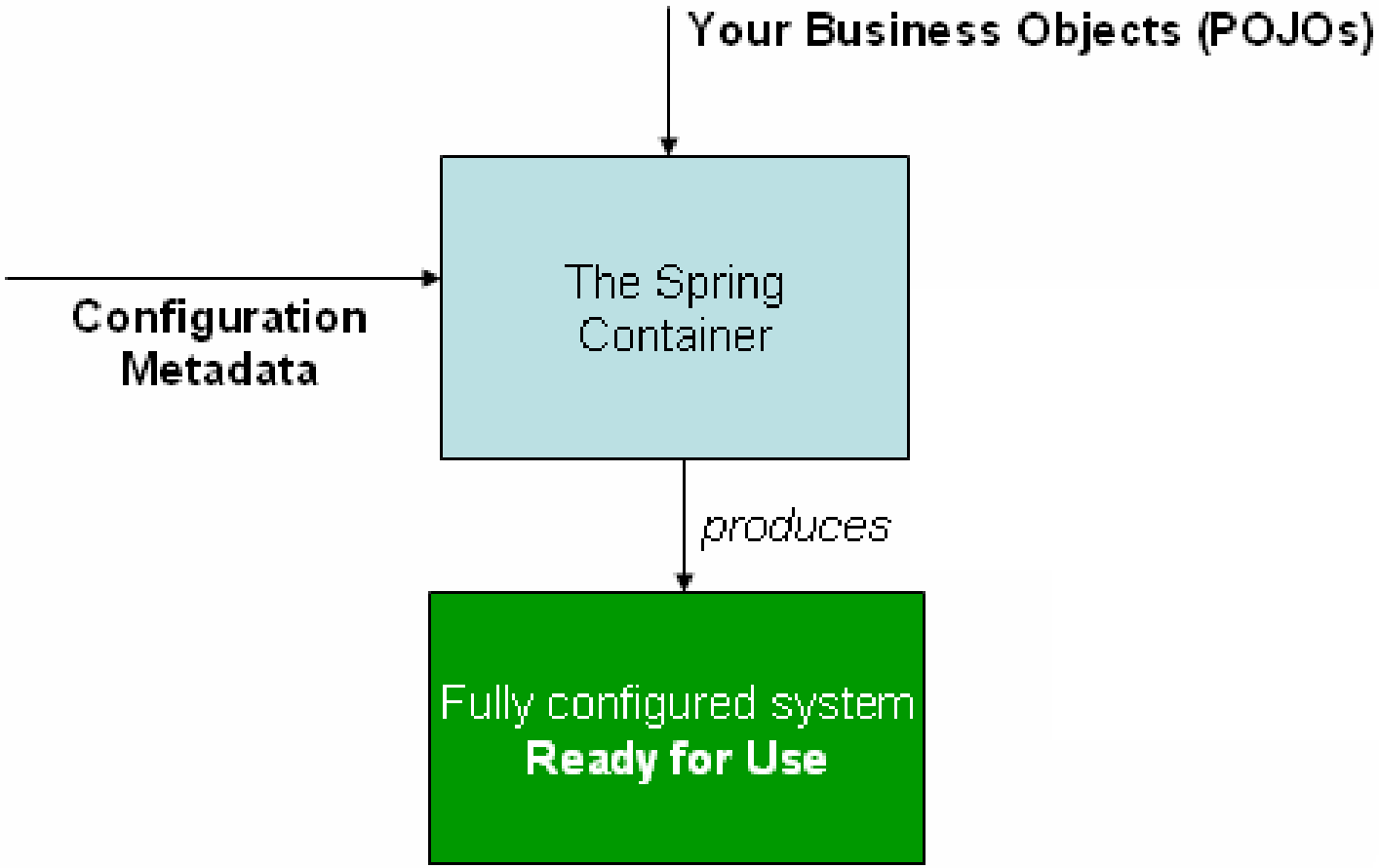
Spring Container

- Responsible for:
  - Creating, configuring and wiring beans together
  - A “bean” is a Java class that the container manages
- Metadata is read from an XML configuration file
  - Configuration file may direct the container to scan the code for additional metadata contained in Java annotations
- The container is made up of
  - A bean factory that manages the beans
  - A Spring context that provides additional functionality
    - *The context used depends on the application type*
    - *Application contexts are for stand alone applications*
    - *Web contexts are for web-based applications*

Spring Container

- Responsible for:
  - Creating, configuring and wiring beans together
  - A “bean” is a Java class that the container manages
- Metadata is read from an XML configuration file
  - Configuration file may direct the container to scan the code for additional metadata contained in Java annotations
- The container is made up of
  - A bean factory that manages the beans
  - A Spring context that provides additional functionality
    - *The context used depends on the application type*
    - *Application contexts are for stand alone applications*
    - *Web contexts are for web-based applications*

Spring Container



Spring Beans

- A Spring bean is an instance of a POJO created the container that has been configured using the supplied metadata
- A “Bean Definition” consists of:
  - The class name for the POJO to be managed
  - An interface that the bean implements
  - The scope and lifecycle of the bean
  - Reference to other beans needed to satisfy dependencies
  - Runtime settings like size limits
- Beans have a unique name
  - Can be defined in the metadata
  - Defaults to the name of the Java class
  - Beans are a wrapper around a POJO (Decorator pattern)
  - The bean implements the same interface as the POJO

The Spring Framework

- Organized into a series of modules
  - Each provides a specific sort functionality
  - In the second week you saw the JPA data module at work
- The core container is the heart of the Spring Framework
  - It manages all the different services
  - Wires them together to form a larger application
  - Manages the low level operations like Java object creating and lifecycles
  - This will become clear in the lab

Creating a Container

- A container is created by creating a specific context
- In the example below, an applictionContext is created
  - Similar sort of idea as an Executor interface that manages thread objects
- In the scope of the context, Spring manages the define beans
  - Once the context is closed, Spring releases the managed resources

```
public static void main(String[] args) {
ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("config.xml");
// working with beans
context.close();
```

Defining a Named Bean

- The code below defines a bean named “SpinDoctor” using the PRWhiz Java class.

```
@Component("SpinDoctor")
public class PRWhiz implements Consultant {

    @Override
    public String getAdvice() {
        return "Don't let them see you sweat";
    }
}
```

Default Name for a Bean

- The code below defines a bean named “ITGuru” by using the Java class name by default

```
@Component
public class ITGuru implements Consultant {

    @Override
    public String getAdvice() {
        return "Turn it off and on again";
    }
}
```

Interfaces

- A bean may be implemented by different Java classes
  - For example, different versions of the class
- In order to decouple the client logic from the container internals
  - Beans are referenced in the client code by a Java Interface reference
  - Every class that is used for making a bean must implement an interface known to the container
- This is consistent with programming best practices
- The bean wrapper is an interceptor
  - Takes client messages, does the management overhead
  - Forwards the message to the POJO and returns the result

Getting a Bean

- We get a reference to a bean by asking the container for a bean by referencing a specific name
  - It may provide a reference to an existing instance
  - It may create a new instance

```
public static void main(String[] args) {
ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("config.xml");


Consultant dilbert = context.getBean("ITGuru",Consultant.class);
Consultant ratbert =  context.getBean("SpinDoctor",Consultant.class);
```

Lab 6-1

Spring Framework

Bean Lifecycle

- By default, a bean is a singleton
  - That means if we ask for a reference to a bean and an instance already exists, we get a reference to that bean
  - At any time, there exists only one copy of the bean
  - One reason for this is to avoid creating multiple copies of a resource during dependency resolution – use existing resources instead
- We can specify other lifecycles
  - If the “Prototype” scope is declared, ever request for a reference to a bean creates a new instance of the bean
  - The discussion of other aspects of lifecycle management is beyond the scope of the course

Bean Scope

- In this definition, the @Scope() annotation is used to ensure a new copy of the bean is created each time a bean is requested.

```
@Component
@Scope("prototype")
public class ITGuru implements Consultant {

    @Override
    public String getAdvice() {
        return "Turn it off and on again";
    }
}
```


Lab 6-2
Bean Life Cycles

Dependency Injection

- A bean may be dependent on other beans
  - This dependencies are identified by the @Autowired annotation
  - Once dependence is identified by Spring, it scans for a bean that implements the interface in the dependency
  - One a bean is instantiated, Spring either creates or gets a reference to an existing bean that satisfies that dependency
- Two common kinds of DI
  - Constructor injection: the dependency is identified in the constructor and resolved when the Java object is created
  - Setter injection: The dependency is identified in a setter method so allow for a looser coupling between instances

Constructor Injection

- In this example, we have a “Manual” bean which is used by an ITGuru bean

```
@Component
public class TechGuide implements Manual {

    @Override
    public String lookup() {
        return "Just a sec ... Googling it.";
    }

}
```

Constructor Injection

- In this example, we have a “Manual” bean which is used by an ITGuru bean

  In the ITGuru bean, we tell Spring we need an instance of a bean that implements the “Manual” interface

```
@Component
public class ITGuru implements Consultant {

    private Manual myManual;

    @Autowired
    public ITGuru(Manual m ) {
        this.myManual = m;
    }

    @Override
    public String getAdvice() {
        return this.myManual.lookup();
    }
}
```

Dependency Injection

- The dependency is defined in terms of an interface
  - This decouples the client bean from any specific implementation of a manual
- In this example, only one TechManual object needs to be present that all the ITGuru objects can share
  - When a bean is used in this way, it is often called a service
  - We generally only need a single copy of a service


Lab 6-3
Dependency Injection


Java™