Introduction to

Spring Core

by Michael Silvanovich

What is Spring?

- Spring is a **lightweight**, but at the same time **flexible** and **universal** framework used for development of **Java SE** and **Java EE** applications
- Spring includes several separate projects, so by saying “Spring” people often mean the entire family of projects
- Spring is **open-source** framework

What is Spring?

- **Rod Johnson**

  created Spring in **2002** with the publication of his book **Expert One-on-One J2EE Design and Development without EJB**

- The framework was first released in **June 2003**
- The basic idea behind Spring is **simplification** of traditional approach (at that moment) to designing and development of J2EE applications

How do Spring simplifies things?

- **Lightweight** and **minimally invasive development** with POJOs
  - Lets you **focus on domain problem**
- **Loose coupling** through DI and interface orientation
- **Declarative programming** through aspects and common conventions
- **Eliminating boilerplate code** with aspects and templates

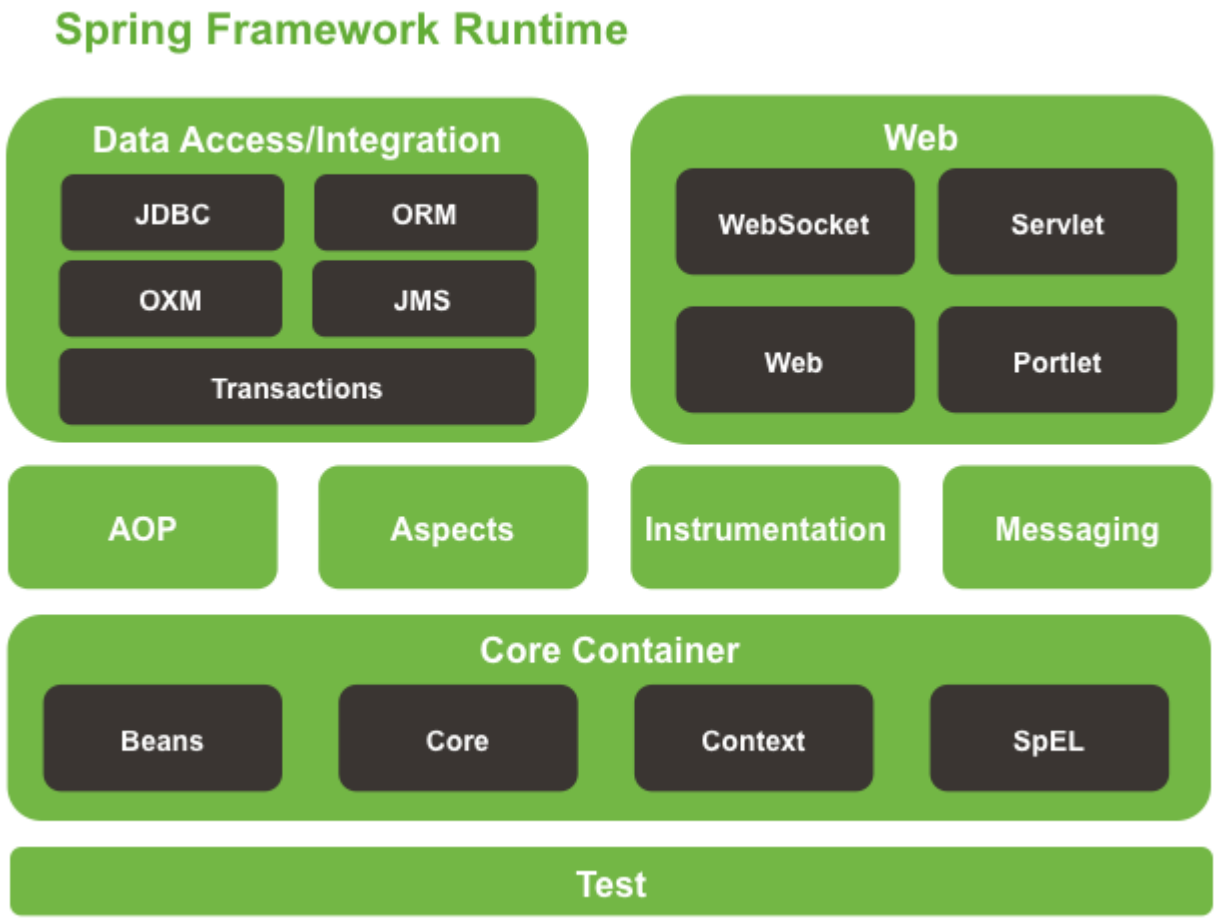
Spring’s design philosophy

- **Provide choice at every level**
- **Accommodate diverse perspectives**
- **Maintain strong backward compatibility**
- **Care about API design**
- **Set high standards for code quality**

Modules

- **Spring Framework**
- **Spring Integration**
- **Spring Batch**
- **Spring Data**
- **Spring Security**
- and many others

Spring Framework



What are we going to talk about?

- We are going to talk about **Spring Core:**
  - **IoC container**
  - **Events (Homework)**
  - **SpEL (Homework)**
  - **AOP**

Introduction

- **Example (what's wrong with this code?)**

```
public class DriverImpl implements Driver {

    private Car car = new CarImpl();

    @Override
    public void drive() {
        this.car.go();
    }

}
```

Introduction

- **Dependency Injection**
  - constructor injection (better, we can test it!)

```
public class DriverImpl implements Driver {

    private Car car;

    public DriverImpl(Car car) {
        this.car = car;
    }

    @Override
    public void drive() {
        this.car.go();
    }

}
```

Introduction

- **Dependency Injection**
  - setter injection (good, as well, our driver now can change a car!)

```
public class DriverImpl implements Driver {

    private Car car;

    public DriverImpl() {}

    @Override
    public void drive() {
        this.car.go();
    }

    public void setCar(Car car) {
        this.car = car;
    }
}
```

Introduction

- **Dependency Injection**
  - constructor injection
  - setter injection
- The process of injecting dependency is called **wiring**
  - **Spring** can do it for you

Introduction

- **Constructor injection**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="car" class="CarImpl" />

    <bean class="DriverImpl">
        <constructor-arg ref="car" />
    </bean>

</beans>
```

Introduction

- **Setter injection**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="car" class="CarImpl" />

    <bean class="DriverImpl">
        <property name="car" ref="car"/>
    </bean>

</beans>
```

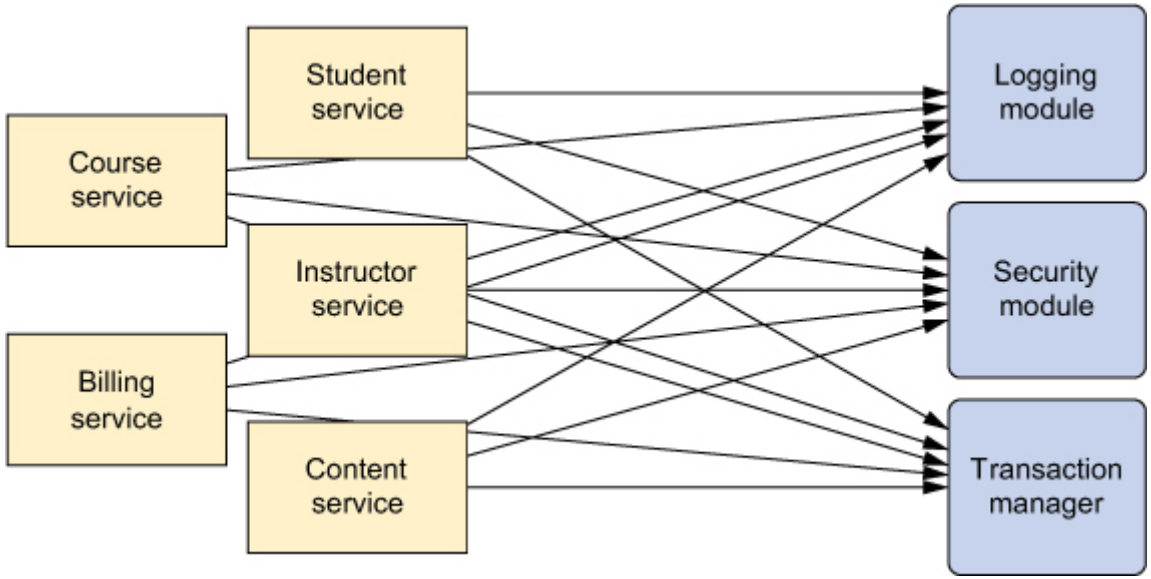
Introduction

- **Code**

```
ApplicationContext context =
                new ClassPathXmlApplicationContext("classpath:/beans.xml");

Driver driver = context.getBean(Driver.class);
driver.drive();
```

Introduction

- **Cross-cutting concerns**



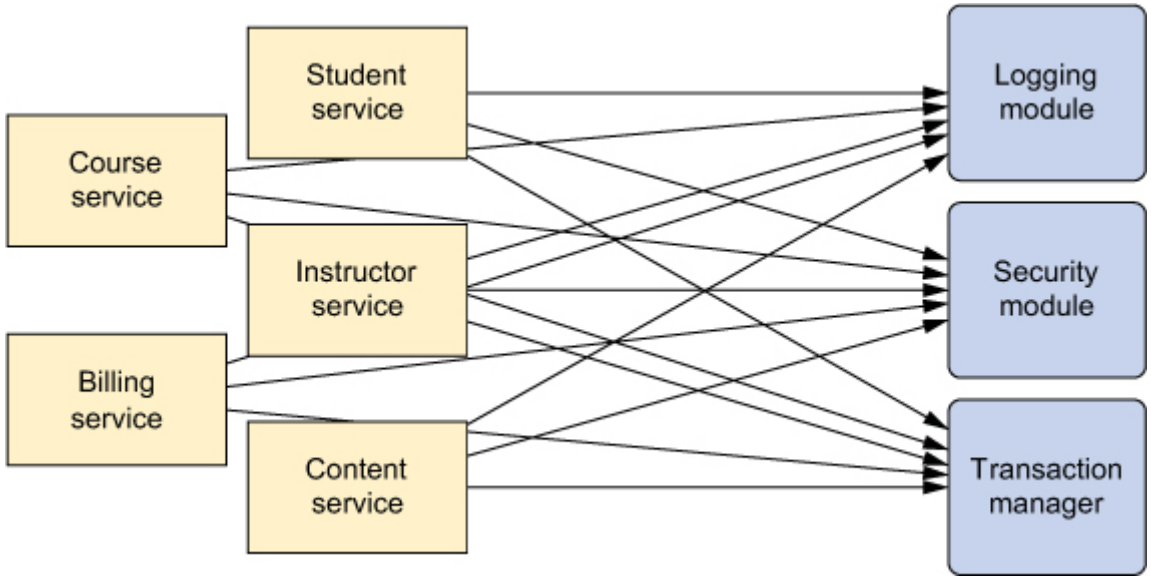
Introduction

- Our driver needs a mechanic to check a car before and after he or she drives a car
- **What's wrong with this code?**

```
public class DriverImpl implements Driver {

    private Car car;
    private Mechanic mechanic;

    public void setCar(Car car) {
        this.car = car;
    }

    public void setMechanic(Mechanic mechanic) {
        this.mechanic = mechanic;
    }

    @Override
    public void drive() {
        this.mechanic.checkBeforeDriving();
        this.car.go();
        this.mechanic.checkAfterDriving();
    }

}
```

Introduction

- **Solution: AOP**
  - **Spring** gives you **AOP**



Introduction

- **There is no need to change DriverImpl**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="...">

    <bean id="car" class="CarImpl" />
    <bean id="mechanic" class="MechanicImpl" />

    <bean class="DriverImpl">
        <property name="car" ref="car"/>
    </bean>

    <aop:config>
        <aop:aspect ref="mechanic">
            <aop:pointcut id="drive" expression="execution(* *.drive(..))"/>

            <aop:before pointcut-ref="drive" method="checkBeforeDriving"/>

            <aop:after pointcut-ref="drive" method="checkAfterDriving"/>
        </aop:aspect>
    </aop:config>

</beans>
```

Introduction

- **Now, that’s nice**

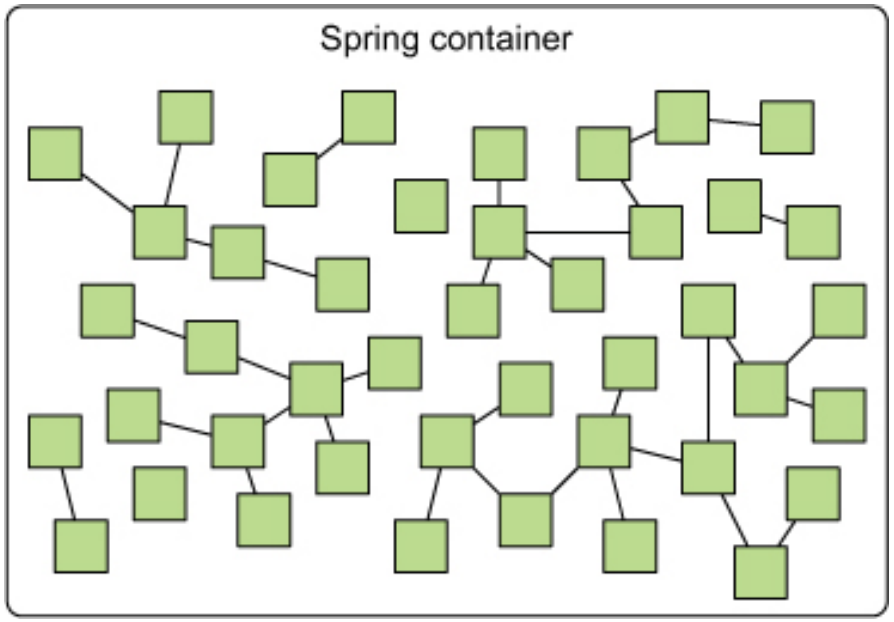
```
public class DriverImpl implements Driver {

    private Car car;

    public DriverImpl() {}

    @Override
    public void drive() {
        this.car.go();
    }

    public void setCar(Car car) {
        this.car = car;
    }
}
```

How do Spring simplifies things?

- **Lightweight** and **minimally invasive development** with POJOs
  - **we used only POJOs**
- **Loose coupling** through DI and interface orientation
  - **we've been injecting dependencies**
- **Declarative programming** through aspects and common conventions
  - **we've applied AOP for separation of concerns**
- **Eliminating boilerplate code** with aspects and template
  - **we've applied AOP for separation of concerns**
  - **Spring uses Templates for boilerplate code elimination, we don't cover it in this presentation**

Spring container

- **Spring container –** is a container where Spring-based application objects live
- **Spring container** knows which objects it should
  - **create**
  - **configure**
  - **wire**
- **Spring container** manages its beans life-cycles



Spring container

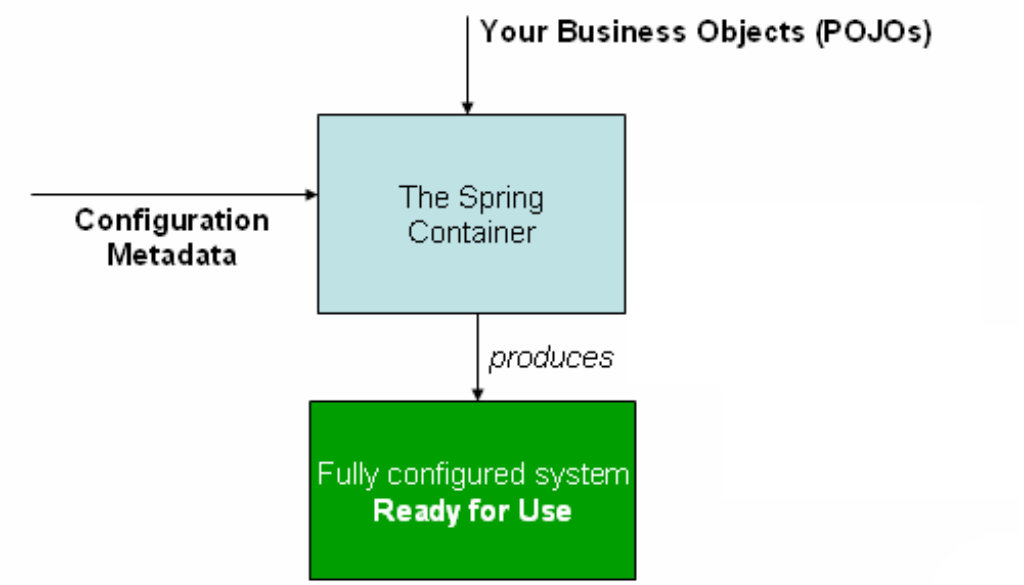
- **Spring container**
  - **BeanFactory**
    - provides basic support for DI
  - **ApplicationContext**
    - support for DI
    - provides application-framework services

Spring container

- **ApplicationContext**
  - AnnotationConfigApplicationContext
  - AnnotationConfigWebApplicationContext
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext

Spring container

- Basically, Spring container's job can be illustrated like this



Spring container

- **AnnotationConfigApplicationContext**

```
ApplicationContext context =
                new AnnotationConfigApplicationContext(JavaConfig.class);
```

- **ClassPathXmlApplicationContext**

```
ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("classpath:/beans.xml");
```

Spring container

- **Getting a bean**

```
Driver driver = applicationContext.getBean(Driver.class);
```

- **What is a bean?**
  - It is just a **POJO** (Plain Old Java Object)

```
public class HelloWorld {

    public String sayHello() {
        return "Hello, World!";
    }

}
```

Bean instantiation

- **Through constructor**

```
<bean class="CarImpl" />
```

- **Through static factory method**

```
<bean id="car" class="CarImpl" factory-method="getInstance"/>
```

- **Through non-static factory method**

```
<bean id="car" factory-bean="carFactory" factory-method="getInstance"/>
```

Dependency injection

- **Through constructor**

```
<bean id="car" class="CarImpl" />

<bean class="DriverImpl">
    <constructor-arg ref="car" />
</bean>
```

- **Through setter**

```
<bean id="car" class="CarImpl" />

<bean class="DriverImpl">
    <property name="car" ref="car"/>
</bean>
```

Autowiring

- **Autowiring** is a feature enabling you to inject object dependency implicitly
  - **no** (default)
  - **byName**
  - **byType**
  - **constructor**
  - **autodetect** (constructor, then byType)

Autowiring

- **byName**
  - **Spring** will be looking for dependency by property name (should be equal to bean ID)

```
<bean id="car" class="CarImpl" />
<bean class="DriverImpl" autowire="byName" />
```

  - **DriverImpl** has a property called “car”
- **byType**
  - **Spring** will be looking for dependencies by property type

```
<bean id="car" class="CarImpl" />
<bean class="DriverImpl" autowire="byName" />
```

  - **DriverImpl** has a property of type **Car** (CarImpl implements Car)
  - **Works if and only if there is only one bean of required type**

Autowiring

- **contructor**
  - **Spring** will be looking for dependencies by constructor argument type

```
<bean class="CarImpl" />
<bean class="DriverImpl" autowire="constructor" />
```

  - **DriverImpl** has a constructor with argument of type **CarImpl** (or type of the same interface)
  - **Works if and only if there is only one bean of required type**
- **autodetect**
  - **Spring** will try to apply **constructor** autowiring, and if it fails then it will try to use **byType**

Autowiring

- **no** (default)
  - means that there will be no autowiring.
- **byName**
  - **Spring** will try to find a bean with the same ID as a property declaring dependency. Left **null** if not found.
- **byType**
  - **Spring** will try to find a bean with the same type, if there is only one. If not found, or too many suitable beans, it will throw **UnsutisfiedDependencyException**.
- **Constructor**
  - **Spring** will try to find beans satisfying constructor argument types. If there is more than one bean satisfying constructor argument type, or there is more than one constructor to try, it will throw **UnsutisfiedDependencyException**.
- **autodetect** (constructor, then byType)

Annotations

- **Spring** supports configuration using annotations
- Common annotation types used
  - **@Required**
  - **@Autowired**
  - **@Component**
  - **@Qualifier**
- Can be enabled by specifying

```
<context:annotation-config />
```

Annotations

- **@Required**
  - applicable only to SETTERS
  - indicates that dependency should be satisfied via configuration or autowiring, throws exception if it isn’t

Annotations

- **@Autowired(required=true|false)**
  - applicable to setters
  - applicable to constructors
  - applicable to setters with more than one parameter
  - applicable to fields (even private)
  - applicable to arrays and typed collections (all beans of collection type will be injected)
- **@Qualifier("name")**
  - applicable **along with** **@Autowired** to specify the exact name of a bean you want to inject

Annotations

- **@Component**
- **@Component("name")**
  - applicable to classes
  - used to avoid xml configuration
  - base stereotype for any spring-managed component: **@Service**, **@Repository, @Controller**
    - **Homework: what is the difference between @Service, @Repository and @Controller?**

Wiring

- How to create discoverable bean?
  - Mark it with annotation **@Component**

```
@Component
public class CarImpl implements Car {

    @Override
    public void go() {
        System.out.println("brum... brum...");
    }

}
```

Wiring

- How to tell Spring I want the discovery?
  - Java configuration

```
@Configuration
@ComponentScan
public class JavaConfig {}
```

  - xml configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="test.scanning" />

</beans>
```

Wiring

- The essense of DI is **wiring**
- **Spring** supports **wiring mechanisms**
  - **XML configuration**
  - **Java configuration**
  - **Bean discovery and automatic wiring**
    - Component scanning
    - Autowiring

Wiring

- **@Configuration** indicates that class declares one or more beans, it means that our class is a configuration class
- **@Component** indicates that class represents bean type and tells container that it's instances will be managed by it
- **@Component**(**`"some_name"`**) can be used to give bean a name. By default decapitalized short class name is used

Component scan

- **@ComponentScan** indicates that we need spring to do bean discovery for us
  - Must be placed only on configuration classes
  - Allows to change component scan behaviour by specifying and include and exclude filters

```
@ComponentScan(
    excludeFilters =
      @ComponentScan.Filter(
          type = FilterType.ASSIGNABLE_TYPE,
          value = RuntimeConfiguration.class
    ),
    includeFilters =
      @ComponentScan.Filter(
          type = FilterType.ASSIGNABLE_TYPE,
          value = ThirdPartyLibraryClass.class
    ),
    basePackageClasses = EntryPoint.class
)
public class ApplicationConfig {
```

Component scan

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {

  @AliasFor("basePackages")
  String[] value() default {};

  @AliasFor("value")
  String[] basePackages() default {};

  Class<?>[] basePackageClasses() default {};

  Class<? extends BeanNameGenerator> nameGenerator() default BeanNameGenerator.class;

  Class<? extends ScopeMetadataResolver> scopeResolver() default AnnotationScopeMetadataResolver.class;

  ScopedProxyMode scopedProxy() default ScopedProxyMode.DEFAULT;

  String resourcePattern() default
                          ClassPathScanningCandidateComponentProvider.DEFAULT_RESOURCE_PATTERN;

  boolean useDefaultFilters() default true;

  Filter[] includeFilters() default {};

  Filter[] excludeFilters() default {};

  boolean lazyInit() default false;

}
```

Component scan

- **basePackages** is an array of package names where to apply component scanning
- **basePackageClasses** is an array of package classes residing in packages where to apply component scanning
- **includeFilters** is an array of filters used to control which types are eligible for component scanning
- **excludeFilters** is an array of filters used to control which types are not eligible for component scanning
- **useDefaultFilters** indicates whether to automatically include types annotated with one of stereotype annotations (@Component, @Service, ...)
- **Homework**: all other options

Component scan

- **@ComponentScan.Filter** is used to configure filtering candidates

```
@Retention(RetentionPolicy.RUNTIME)
@Target({})
public @interface Filter {

    FilterType type() default FilterType.ANNOTATION;

    Class<?>[] value() default {};

    @AliasFor("value")
    Class<?>[] classes() default {};

    String[] pattern() default {};

}
```

Component scan

- **FilterType**
  - ANNOTATION
  - ASSIGNABLE_TYPE
  - ASPECTJ
  - REGEX
  - CUSTOM
    - You will need to implement TypeFilter interface, your class should have no arguments constructor

Mixing configurations

- **@Import({JavaConfig.class}) –** indicates that container should also use definitions from specified java configuration classes
  - applicable to configuration classes annotated with **@Configuration**
- **@ImportResource({"`classpath:/beans.xml`"}) –** indicates that container should also use definitions from specified xml configuration classes
  - applicable to configuration classes annotated with **@Configuration**

Advanced wiring

- **Addressing ambiguity in autowiring**
  - **Problem**

```
@Configuration
static class JavaConfigWithError {

    @Bean
    public TestBean defaultTestBean() {
        return new DefaultTestBean();
    }

    @Bean
    public TestBean otherTestBean() {
        return new OtherTestBean();
    }

}

...

TestBean testBean = context.getBean(TestBean.class);
```

Advanced wiring

- **Addressing ambiguity in autowiring**
  - **@Primary**
    - can be applied to **@Component** annotated type and bean configuration method

```
@Configuration
public class JavaConfigWithNoError {

    @Primary
    @Bean
    public TestBean defaultTestBean() {
        return new DefaultTestBean();
    }

    @Bean
    public TestBean otherTestBean() {
        return new OtherTestBean();
    }

}
```

Advanced wiring

- **Addressing ambiguity in autowiring**
  - **Problem**

```
@Configuration
public class JavaConfigWithError {

    @Bean
    public TestBean defaultTestBean() {
        return new DefaultTestBean();
    }

    @Bean
    public TestBean otherTestBean() {
        return new OtherTestBean();
    }

    @Bean
    @Autowired
    public List<TestBean> testBeanList(TestBean testBean) {
        return Collections.singletonList(testBean);
    }

}
```

Advanced wiring

- **Addressing ambiguity in autowiring**
  - **@Qualifier**
    - can be applied to field or parameter

```
@Configuration
static class JavaConfigWithNoError {

    @Bean
    public TestBean defaultTestBean() { ... }

    @Bean
    public TestBean otherTestBean() { ... }

    @Bean
    @Autowired
    public List<TestBean> testBeanList(@Qualifier("otherTestBean") TestBean testBean) {
        return Collections.singletonList(testBean);
    }

}
```

Advanced wiring

- **@Profile**({**"dev","integration"**})
  - indicates that component is eligible for registration in case one or more specified profiles are active (**dev** and **integration** in the example above)
  - applicable to @**Component** annotated types, configuration classes, and configuration methods annotated with @**Bean**
- Active profiles can be specified:
  - using JVM property **spring.active.profiles** (comma separated)
  - using @ActiveProfiles in integration test
  - **Home work: How else it can be specified?**

Advanced wiring

- **@Profile example**

```
@Configuration
static class IndicatorConfig {

    @Profile(Indicator.DEV)
    @Bean
    public Indicator devIndicator() {
        return new DevIndicator();
    }

    @Profile(Indicator.PROD)
    @Bean
    public Indicator prodIndicator() {
        return new ProdIndicator();
    }

}
```

Advanced wiring

- **@Conditional({ProfileCondition.class})**
  - indicates that component is eligible for registration in case when all specified conditions match
  - applicable to @**Component** annotated types, configuration classes, and configuration methods annotated with @**Bean**

Advanced wiring

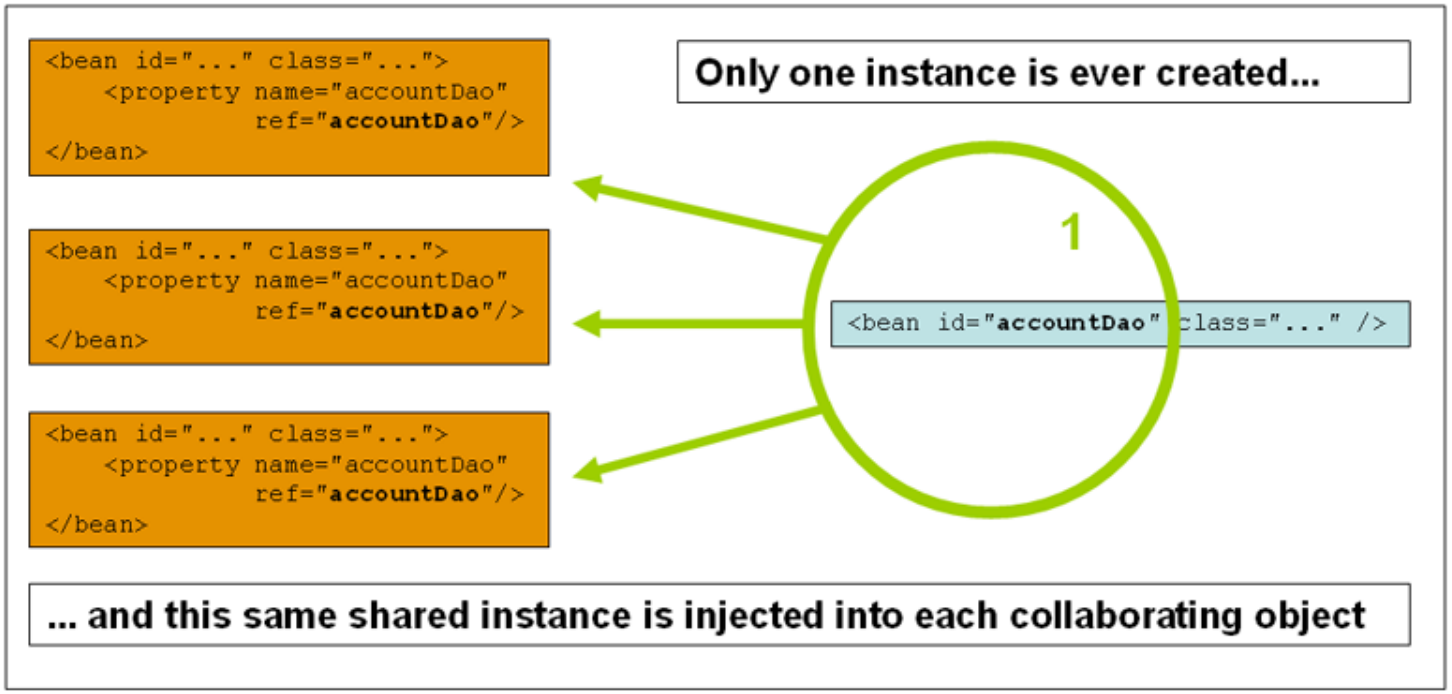
- **@Conditional**

```
@Conditional(MagicExists.class)
@Component
public class WithMagic {}

...

public class MagicExists implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return context.getEnvironment().getProperty("magic") != null;
    }

}
```

- **Homework: what can you get from ConditionContext and AnnotatedMetadata?**

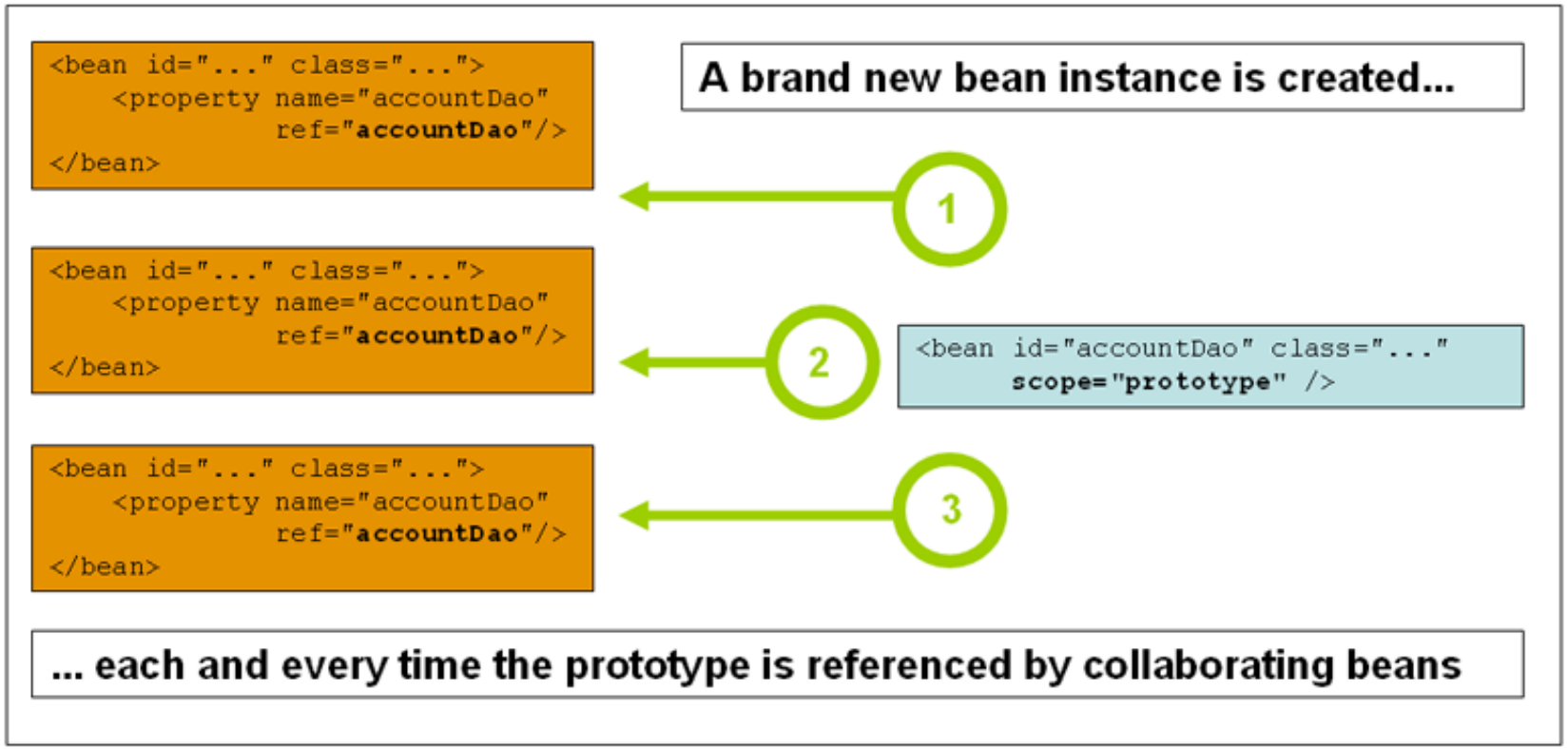
Bean scopes

- **Singleton**
  - default
  - only one instance within container



Bean scopes

- **Prototype**
  - Every call to **getBean**() returns new instance



Bean scopes

- **@Scope**
  - indicates scope to use with instances
  - can be applied to **@Component** annotated types, and bean configuration methods (**@Bean**)

```
@Configuration
public class MqThreadPoolConfiguration {

    @Scope("prototype")
    @Bean(destroyMethod = "destroy", name = "inboundMQAsyncExecutor")
    public TaskExecutorFactoryBean inboundMQAsyncExecutor() {
        final TaskExecutorFactoryBean result = new TaskExecutorFactoryBean();
        result.setQueueCapacity(Integer.MAX_VALUE);
        result.setPoolSize(Application.DEFAULT_CONCURRENCY_LEVEL + "-" +
Application.DEFAULT_CONCURRENCY_LEVEL);
        return result;
    }

...

}
```

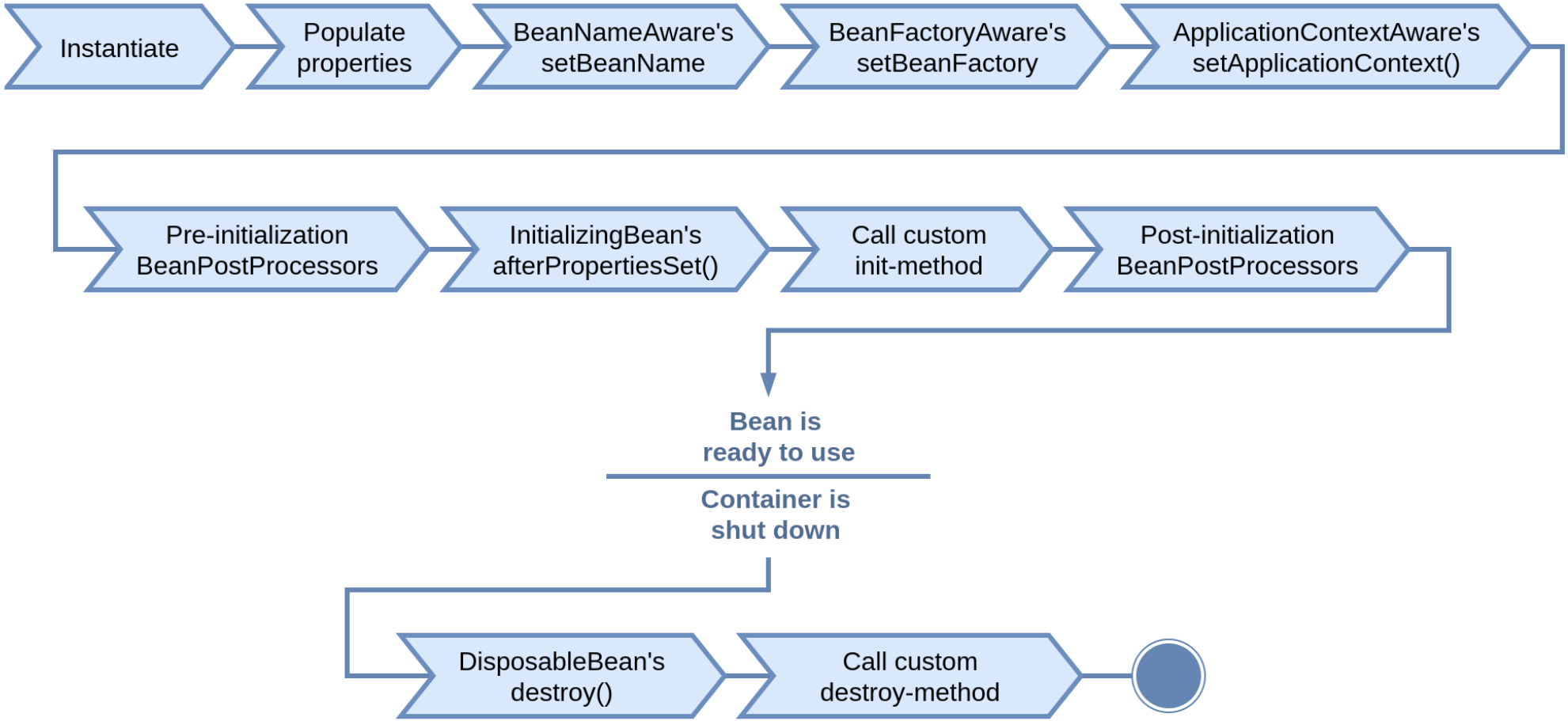
Bean scopes

- **@Scope**
  - **Homework**: **@Scope** has method called **proxyMode()** what does it mean?

Bean scopes

- **Homework:** there are other bean scopes
  - **request**
  - **session**
  - **globalSession**
  - **application**
  - **Websocket**
- **Homework: scoped proxy**

A bean’s life


Instantiate
Populate properties
BeanNameAware's setBeanName
BeanFactoryAware's setBeanFactory
ApplicationContextAware's setApplicationContext()
Pre-initialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is ready to use
Container is shut down
DisposableBean's destroy()
Call custom destroy-method

Callbacks

- **BeanNameAware**

```
public class NamedBean implements BeanNameAware {

    private String beanName;

    @Override
    public void setBeanName(String name) {
        this.beanName = name;
    }

}
```

- **BeanFactoryAware**

```
public class TestBean implements BeanFactoryAware {

    private BeanFactory beanFactory;

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = beanFactory;
    }
}
```

Callbacks

- **ApplicationContextAware**

```
public class TestBean implements ApplicationContextAware {

  private ApplicationContext applicationContext;

  @Override
  public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
      this.applicationContext = applicationContext;
  }
}
```

Callbacks

- **InitializingBean**

```
public class TestBean implements InitializingBean {

    private boolean initialized;

    @Override
    public void afterPropertiesSet() {
        this.initialized = true;
    }

}
```

- **JSR-250**

```
public class TestBean {

    private boolean initialized;

    @PostConstruct
    public void afterPropertiesSet() {
        this.initialized = true;
    }

}
```

Callbacks

- **Init and destroy methods**

```
@Configuration
public class DataSourceConfig {

    @Bean(initMethod = "init", destroyMethod = "close")
    public DataSource dataSource() {
        return new HikariDataSource();
    }

}
```

Callbacks

- **Lifecycle**

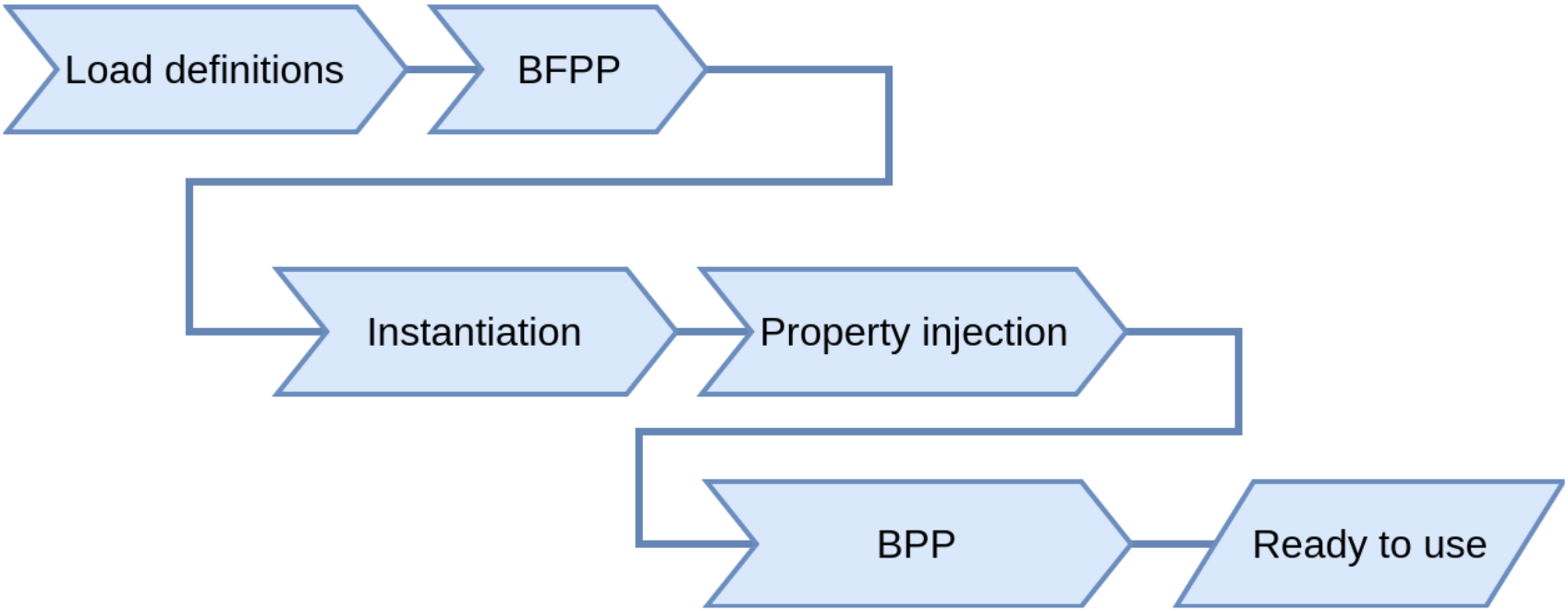
```
public class TestBean implements Lifecycle {

    private final Object lifecycleMonitor = new Object();
    private boolean running;

    @Override
    public void start() {
        synchronized (this.lifecycleMonitor) {
            this.running = true;
        }
    }

    @Override
    public void stop() {
        synchronized (this.lifecycleMonitor) {
            this.running = false;
        }
    }

    @Override
    public boolean isRunning() {
        synchronized (this.lifecycleMonitor) {
            return this.running;
        }
    }
}
```

Callbacks

- **SmartLifecycle (Graceful shutdown)**

```
public class TestBean implements SmartLifecycle {

    private final Object lifecycleMonitor = new Object();
    private boolean running;

    @Override
    public void start() { ... }

    @Override
    public void stop() { ... }

    @Override
    public boolean isRunning() { ... }

    @Override
    public boolean isAutoStartup() {
        return true;
    }

    @Override
    public void stop(Runnable callback) {
        stop();
        callback.run();
    }

    @Override
    public int getPhase() {
        return Integer.MAX_VALUE; // will be the near last component to start
    }
}
```

A container’s life



Container extension points

- **BeanPostProcessor**

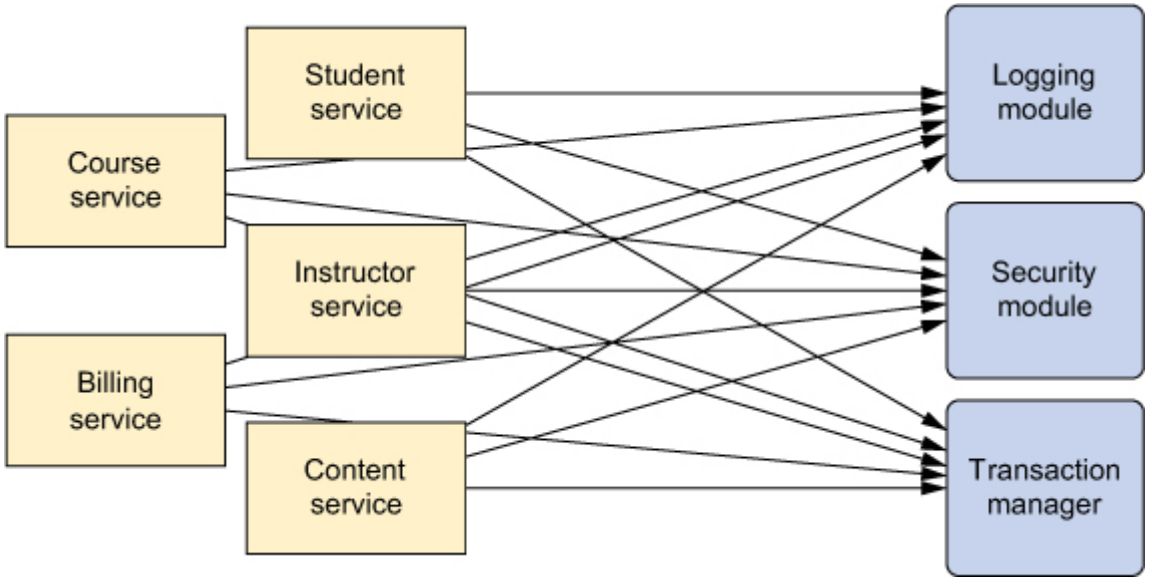
```
@Order(Ordered.HIGHEST_PRECEDENCE)
public class RandomBeanPostProcessor implements BeanPostProcessor {

    private final Random random = new Random();

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        ReflectionUtils.doWithFields(bean.getClass(), field -> {

            if (field.isAnnotationPresent(RandomValue.class)) {
                if (field.getType() == Integer.class || field.getType() == int.class) {

                    if (!field.isAccessible()) field.setAccessible(true);

                    RandomValue value = field.getAnnotation(RandomValue.class);
                    int left = value.leftBound(), right = value.rightBound();
                    int result = random.nextInt(right - left + 1) + left;
                    field.set(bean, result);

                }
            }

        });
        return bean;
    }

}
```

Container extension points

- **Homework**
  - **BeanFactoryPostProcessor**
  - **BeanDefinitionRegistryPostProcessor**

AOP

- **Cross-cutting concerns**
  - **Spring** allows you to modularize it to special classes called **aspects**



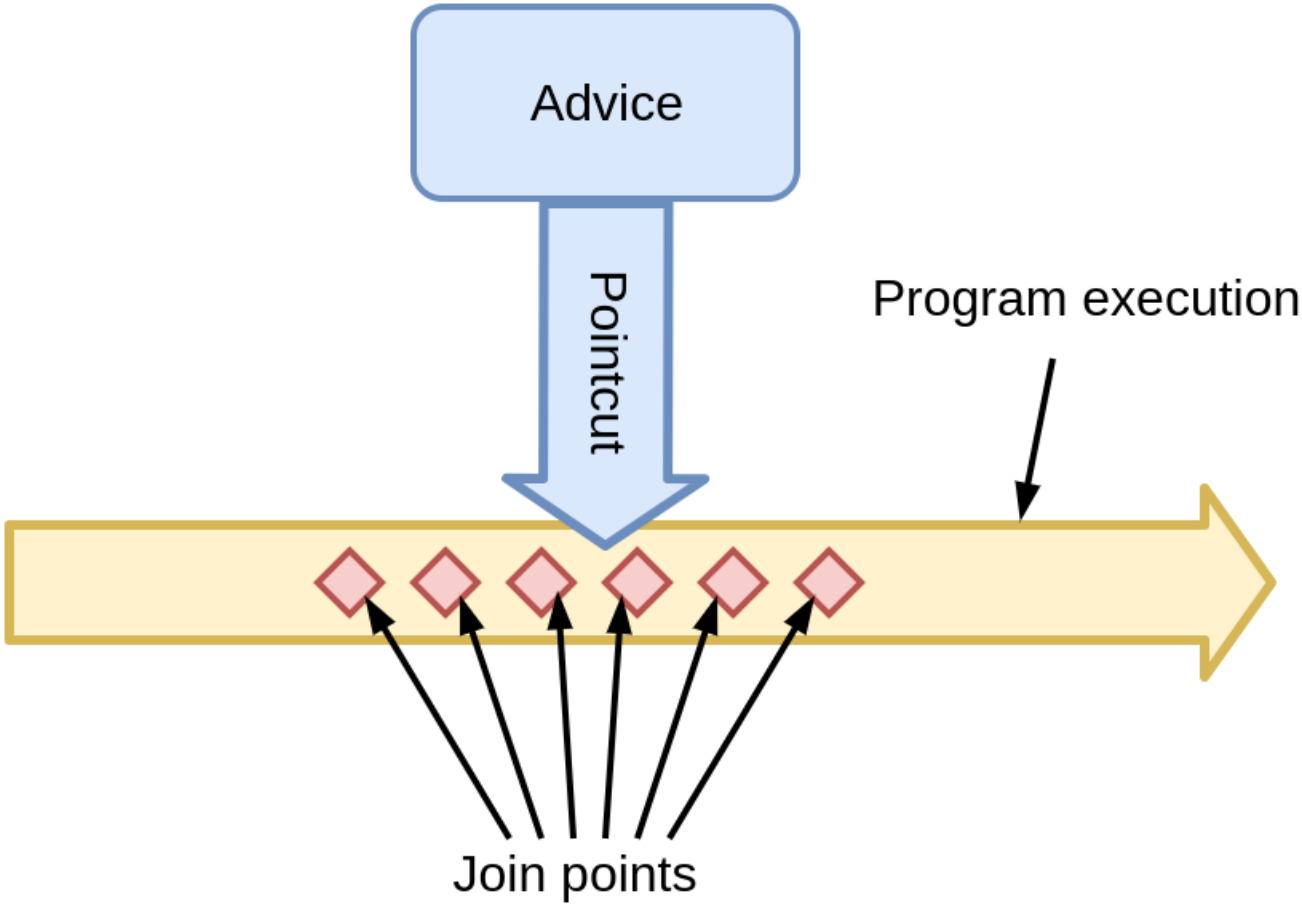
AOP

- **Advice** – the job of an aspect
  - defines both **what** and **when**
    - **before**
    - **after**
    - **after-returning**
    - **after-throwing**
    - **around**

AOP

- **Join point** – a place in execution where aspect can be plugged in (method being called, exception being thrown, field being changed)
- **Pointcut** – defines where to apply aspect (consider it as a filter of join points)
- **Aspect = Advice + Pointcuts**
- **Introduction** – aspect extending interface by adding new methods
- **Weaving** – a processing of applying aspects to a target object

AOP

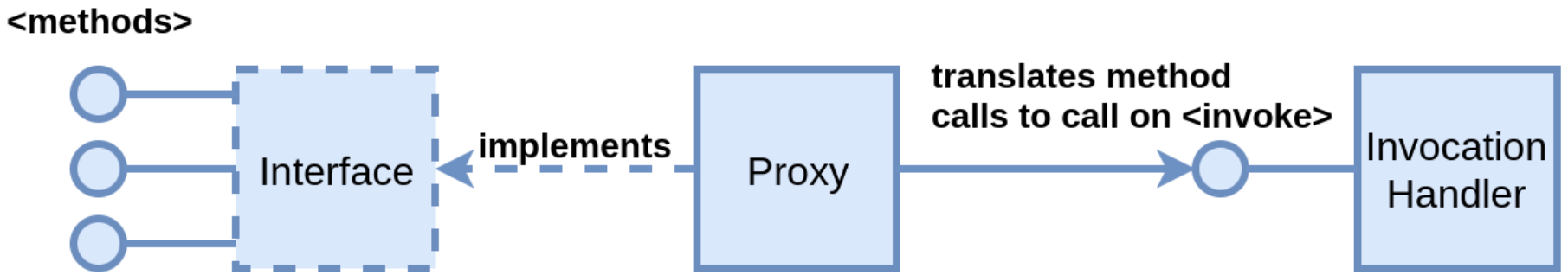


AOP

- **Spring** supports **AOP** in 4 styles:
  - Classic Spring proxy-based **AOP**
  - Pure-POJO aspects
  - @**AspectJ** annotation-driven aspects
  - Injected **AspectJ** aspects
- **Spring AOP** is built around **dynamic proxies**
  - Limited to method interception
  - Java dynamic proxies
  - CGLIB-generated proxies (***optional*** **homework**)
- Injected **AspectJ** aspects are not limited to method interception

Java Dynamic Proxies

- **Java Dynamic Proxy** is a **runtime** generated class, implementing one or more interfaces, that automatically converts every method call to one of those interfaces into a call to the single invoke method on provided implementation of **`java.runtime.InvocationHandler`**:



Java Dynamic Proxies

- How to create a **proxy**?
  - `Object` **`java.lang.reflect.Proxy.`**`newProxyInstance(`

```
ClassLoader classLoader,
Class<?>[] interfaces,
InvocationHandler handler
```

)

    - returns an instance of proxy for the specified interfaces that dispatches method invocations to specified invocation handler
    - ClassLoader **classloader** – classloader aware of all specified interfaces
    - `Class<?>[]` **`interfaces`** – array of interfaces for the proxy class to implement (in implementation order)
    - `InvocactionHandler` **`handler`** – invocation handler to dispatch method invocation to

Java Dynamic Proxies

- What is **InvocationHandler**?
  - **`java.runtime.InvocationHandler`**:

```
public interface InvocationHandler {

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

    - method **invoke** is called instead of original method
    - `Method` **`method`** parameter represents class's method which is called
    - `Object[]` **`args`** is an array of method arguments
    - `Object` **`proxy`** is a reference to a proxy on which method is originally called

Java Dynamic Proxies

Example

```
public <T> T timed(Class<T> iface, T service) {

    return (T) Proxy.newProxyInstance(
            iface.getClassLoader(),
            new Class[]{interfaceClass},
            new InvocationHandler() {

                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

                    if (getServiceMethod(service, method).isAnnotationPresent(Timed.class)) {

                        Timer.Context context = timer.start();
                        try {
                            return method.invoke(service, args);
                        } finally {
                            context.stop();
                            LOG.log("invocation of method " + method.getName()
                                    + " took " + context.elapsed() + " ms.");
                        }

                    } else {
                        return method.invoke(service, args);
                    }
                }
                ...
            }
    );

}
```

Java Dynamic Proxies

Example (usage)

```
interface Service {

    void process(List<Data> data);
}

class ServiceImpl implements Service {

    @Timed
    @Override
    public void process(List<Data> data) {
        Consumer<Data> validateConsumer = this::validate;
        data.forEach(validateConsumer.andThen(this::store));
    }

    void validate(Data data) { /*...*/ }
    void store(Data data) { /*...*/ }

}

...

    Service original = ...
    Service proxied = timed(Service.class, original);

    proxied.process(Collections.emptyList());
```

Java Dynamic Proxies

Example (output)

- `invocation of method process took 35 ms.`

Java Dynamic Proxies

Generated class

- Implements given interfaces
  - All non-public interfaces MUST be in the same package
  - All interfaces MUST be visible by specified class loader
  - Up to 65535 interfaces
- Dispatches method calls to specified invocation handler
  - Can’t proxy static methods
  - Can proxy default methods
- Extends **java.lang.reflect.Proxy**
- **Public**
- **Final**

Java Dynamic Proxies

- **`Proxy#isProxyClass(Class<?> cl)`**
  - returns true if given class is a proxy created by **Proxy.newInstance(...)**
- **`Proxy#getInvocationHandler(Object proxy)`**
  - returns invocation handler assigned to given proxy object
- **`Proxy#getProxyClass(ClassLoader cl, Class<?>… ifaces)`**
  - returns a proxy class instance for given interfaces
  - resulting class can be used to create proxy instances dynamically (it has one argument constructor to use with InvocationHandler)

AOP

- **Annotation-driven AOP**

```
@Aspect
public class MetricsAspect {

    @Autowired private MetricRegistry metricRegistry;

    @Around("@annotation(metered)")
    public Object metricAdvice(ProceedingJoinPoint proceedingJoinPoint,
                               Metered metered) throws Throwable {
        try {
            return proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            metricRegistry.meter(metered.errorMetricName()).mark();
            throw e;
        } finally {
            metricRegistry.meter(metered.value()).mark();
        }
    }

}
```

AOP

- **@Aspect** – indicates that underlying type represents and aspect
  - **value()** - per clause expression for aspect instantiation model
    - **singleton** (default)
    - **perthis (*optional* homework)**
    - **pertarget (*optional* homework)**
- **@Around** – represents around advice
  - applicable to methods only
  - **value()** - represents pointcut expression written in special DSL
  - **argNames()** - sometimes it is required to pass argument names

AOP

- **Annotation-driven AOP**

```
@Aspect
public class MechanicAdvice {

    @Autowired private Mechanic mechanic;

    @Before("execution(* *.drive(..))")
    public void checkBeforeDriving() {
        mechanic.checkBeforeDriving();
    }

    @After("execution(* *.drive(..))")
    public void checkAfterDriving() {
        mechanic.checkAfterDriving();
    }

}
```

AOP

- **@After** – represents after advice
- **@Before** – represents before advice
- **@AfterReturning** – represents after-returning advice
- **@AfterThrowing** – represents after-throwing advice

AOP

- @Pointcut

```
public interface PointcutDefinition {

    @Pointcut("execution(* ru..AccountService.*(..))")
    default void accountService() {};

}

@Aspect
public class ErrorLoggingAdvice implements PointcutDefinition {

    @Autowired private Logger errorLogger;

    @AfterThrowing(
        value = "accountService() && args(account,..)",
        throwing = "ex"
    )
    public void afterThrowing(Account account, Throwable ex) {
        errorLogger.error(
            String.format("Error adjusting balance for account [%s]: %s",
                          account.getCba(), ex.getMessage())
        );
    }

}
```

AOP

- But how to tell Spring to start using aspects?

```
@EnableAspectJAutoProxy
@Configuration
public class Config { ... }
```

- Or with XML

```
<aop:aspectj-autoproxy />
```

AOP

- **Homework**
  - **XML configuration (*optional*)**
  - **Pointcut expression DSL**
  - **Introduction aspect (@DeclareParents)**
  - **AspectJ injected aspects (*optional*)**

Recommend readings

- **Official documentation**
- **Spring in Action**, Craig Walls
- **Expert One-on-One J2EE Design and Development without EJB**, Rod Johnson
- **AspectJ in Action**, Remnivas Laddad

THANK YOU!