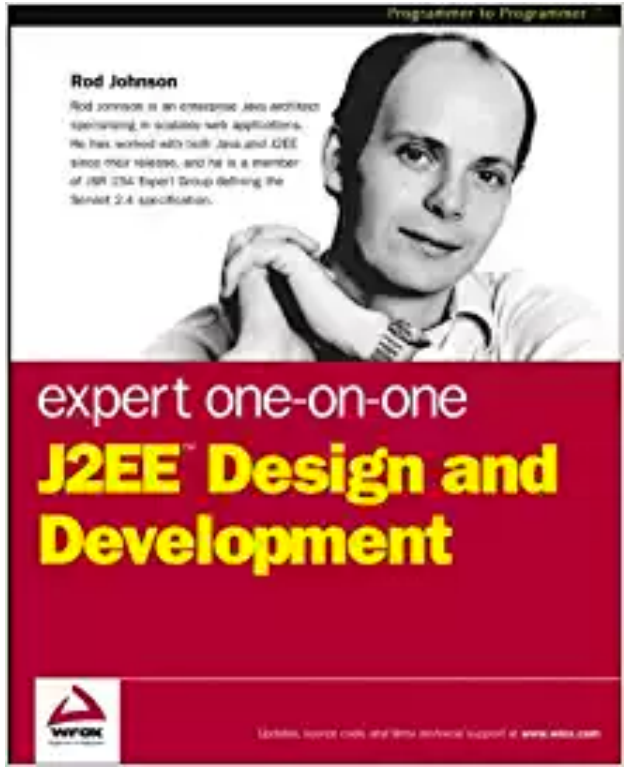
Walmart

Mike Barlotta

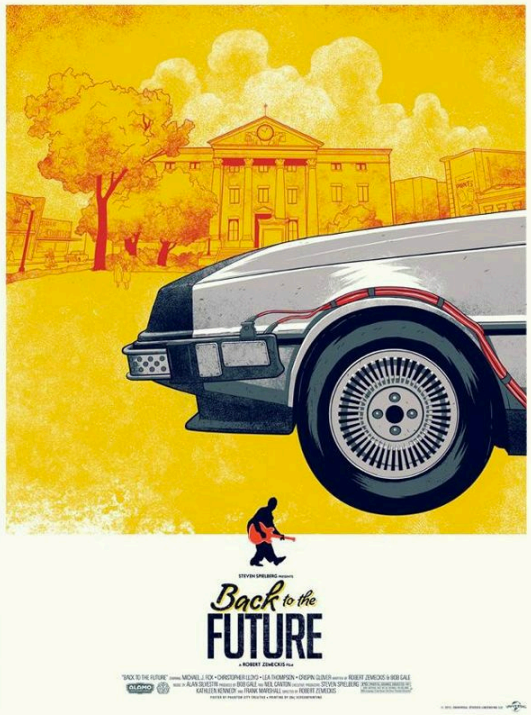
Spring Framework Part 3

What is Spring?

- Dependency Injection framework
  - introduced in 2002 in Rod Johnson's book
  - open sourced in 2003
  - Spring 1.0 available in 2004
- Ecosystem of Capabilities
  - Spring MVC
  - Spring Data
  - Spring Batch
  - Spring Integration
  - Spring Cloud



Where we have been



What is a IoC/DI container & why use them?

Where we have been

Decides when to **create** objects & which type of dependency the client object will **use**

What is a IoC/DI container & why use them?

BeanFactory vs ApplicationContext & configuration options

Where we are going

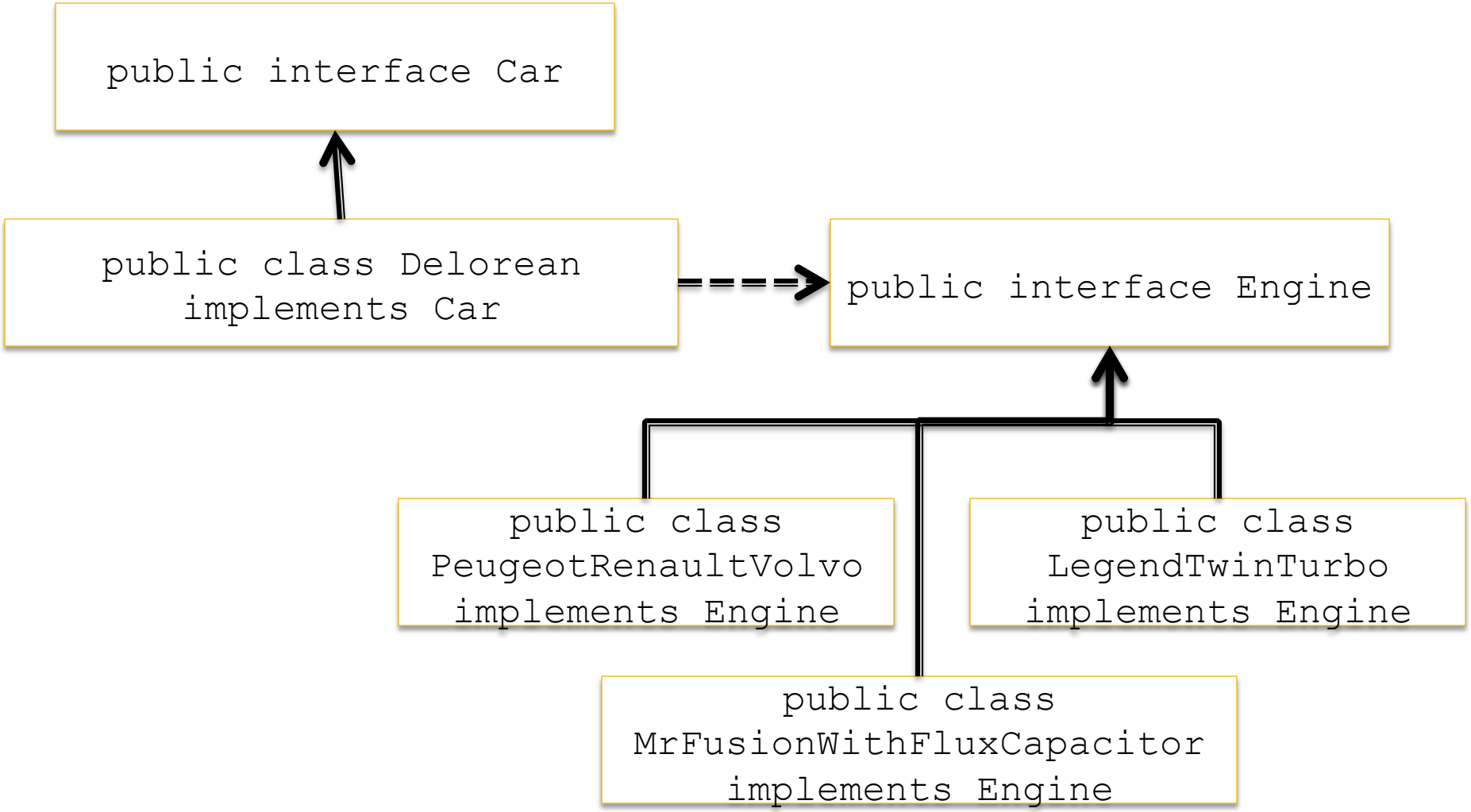
Decides the **number** of instances & **lifecycle** of the client object and dependent objects

What is a IoC/DI container & why use them?

BeanFactory vs ApplicationContext & configuration options

Using Delorean and Engine Design

DI + DIP


public interface Car
public class Delorean implements Car
public interface Engine
public class PeugeotRenaultVolvo implements Engine
public class LegendTwinTurbo implements Engine
public class MrFusionWithFluxCapacitor implements Engine

How many instances should be created by the Spring container?

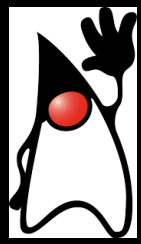
Scope: Singleton

- Default scope for all beans is singleton.
  - Only one instance of the bean is created by the container (*for each bean name*) & shared

Scope: Prototype

- Container creates more than one instance
  - New instance of the bean is created by the container each time one is requested (via call to `getBean` or injection point)

Mixing Scopes



What happens when a prototype bean is injected into a singleton bean instance?

What happens when a singleton bean is injected into a prototype bean instance?

What happens to that bean when Spring build it?

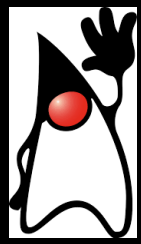
Customizing Initialization & Destruction of a Bean

```
public class Delorean
    implements Car
    InitializingBean,
    BeanNameAware,
    ApplicationContextAware,
    DisposableBean
```

```
@Configuration
public class DeloreanConfig {

    @Bean(name="delorean",
    initMethod = "customInit",
    destroyMethod = "customDestroy")
    @Scope(value = "singleton")
    public Car buildDelorean(Engine engine) {
        …
```

Getting that Bean ready to use



1. Create the Bean
2. Inject the dependencies
3. BeanNameAware interface
4. BeanFactoryAware interface
5. ApplicationContextAware interface
6. --
7. @PostConstruct
8. InitializingBean interface
9. custom init method via @Bean(initMethod)
10. --
11. @PreDestroy
12. DisposableBean interface
13. Custom destory method via @Bean(destroyMethod)

Scope and the Bean's lifecycle

Singleton Scope

- Since there is only one instance the lifecycle is only processed once for the bean

Prototype Scope

- Each instance of the bean goes through the lifecycle
- *No destruction phase of the lifecycle*

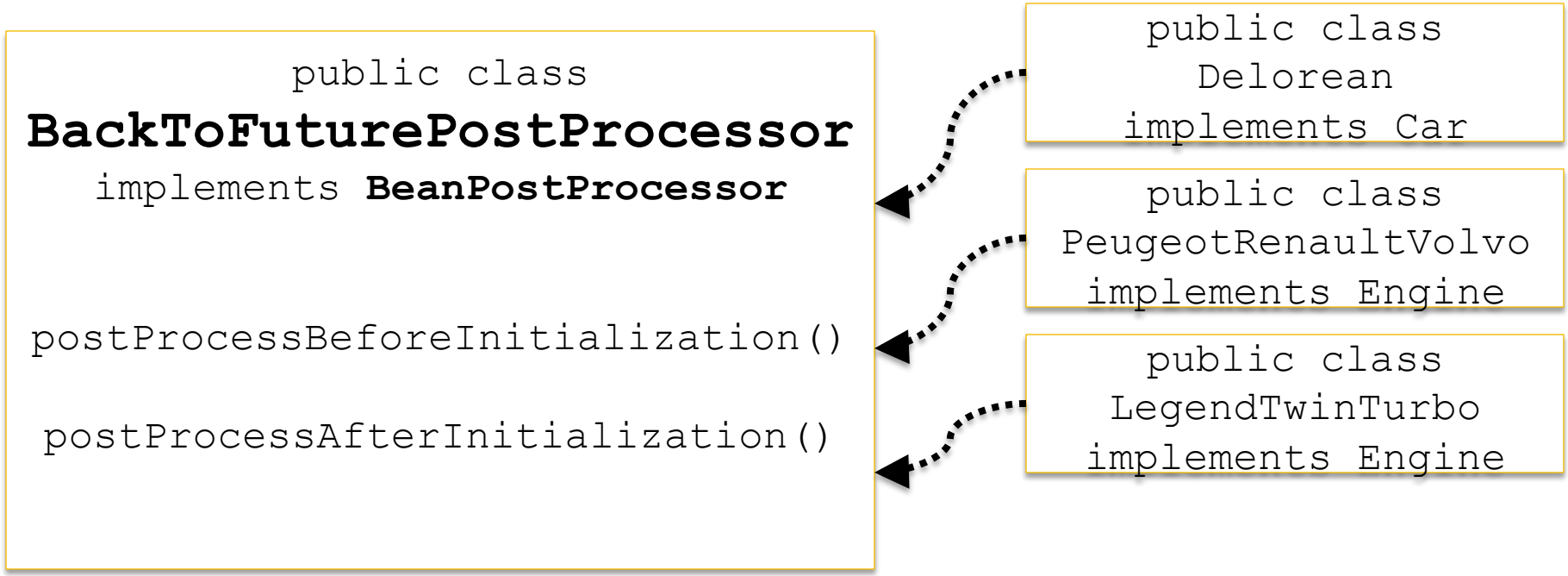
BeanPostProcessor What are they for?

- `BeanPostProcessor` beans are commonly used to extend Spring container.
- Examples in Spring
  - `CommonAnnotationBeanPostProcessor` handles the `@PostConstruct` and `@PreDestroy` annotations on beans
  - `RequiredAnnotationBeanPostProcessor` handles the `@Require` annotation on beans

BeanPostProcessor interface

- An `ApplicationContext` automatically detects any beans in `@Configuration` that implement the interface.
- `BeanPostProcessor` beans and their dependencies are instantiated *before* all other beans. But it can't use `@Value`
- `BeanPostProcessor` beans are used to process all other beans in the container.

Adding BeanPostProcessor(s)



- `BeanPostProcessor` beans are used to process *all* other beans in the container.

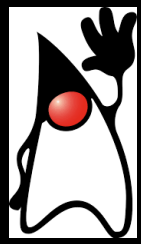
ApplicationContext registers the BPP…

```
appContext = new
AnnotationConfigApplicationContext();

appContext.register(DeloreanConfig.class,…);
appContext.refresh();
```

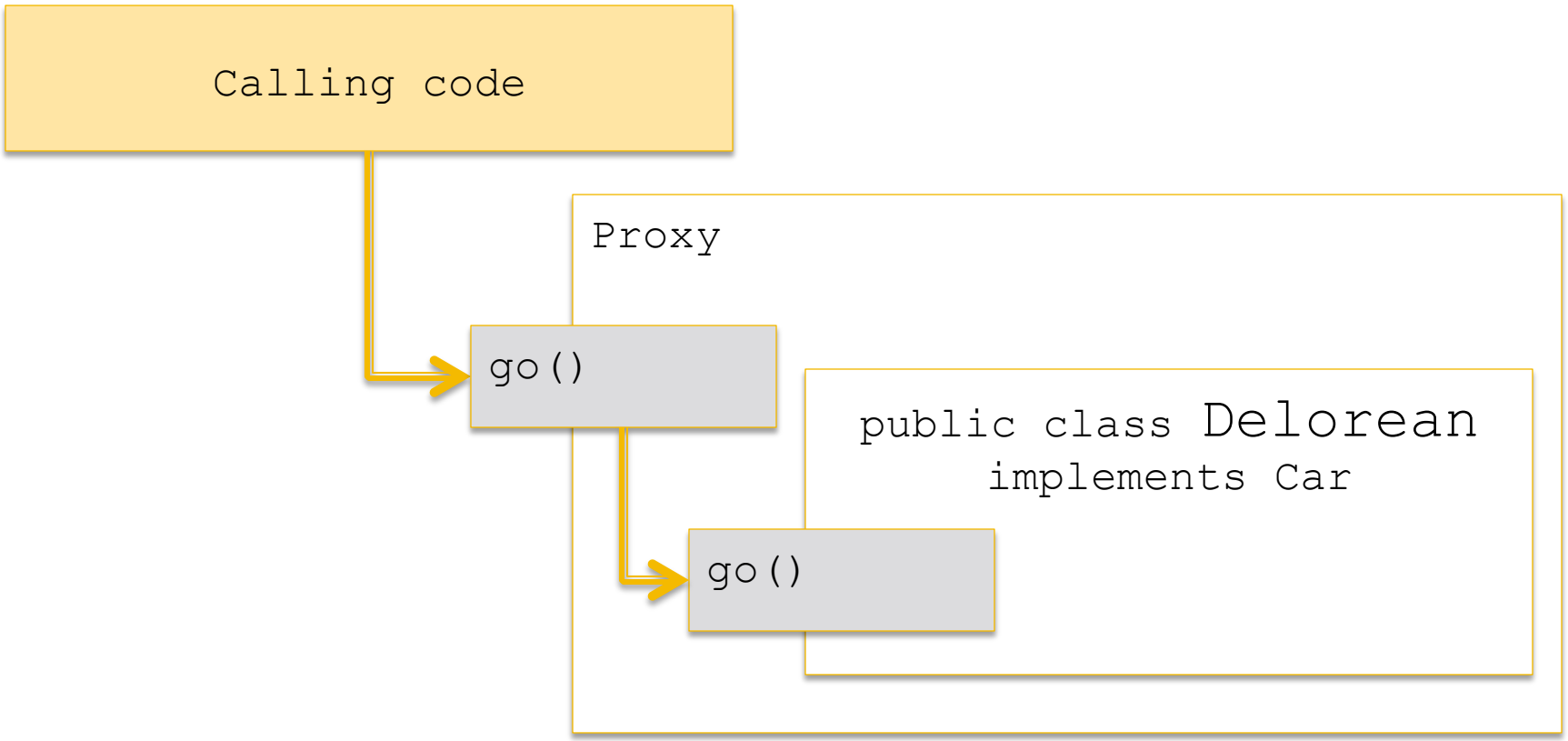
```
public class AbstractApplicationContext …
@Override
public void refresh() … {
    …
    registerBeanPostProcessors(beanFactory);
```

Where do BPP fit in ?



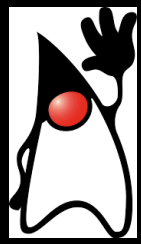
1. Create the Bean
2. Inject dependencies
3. BeanNameAware interface
4. BeanFactoryAware interface
5. ApplicationContextAware interface
6. **BeanPostProcessor.postProcessBeforeInitialization**
7. @PostConstruct
8. InitializingBean interface
9. custom init method via @Bean(initMethod)
10. **BeanPostProcessor.postProcessAfterInitialization**
11. @PreDestroy
12. DisposableBean interface
13. Custom destory method via @Bean(destroyMethod)

Timing the Car.go method

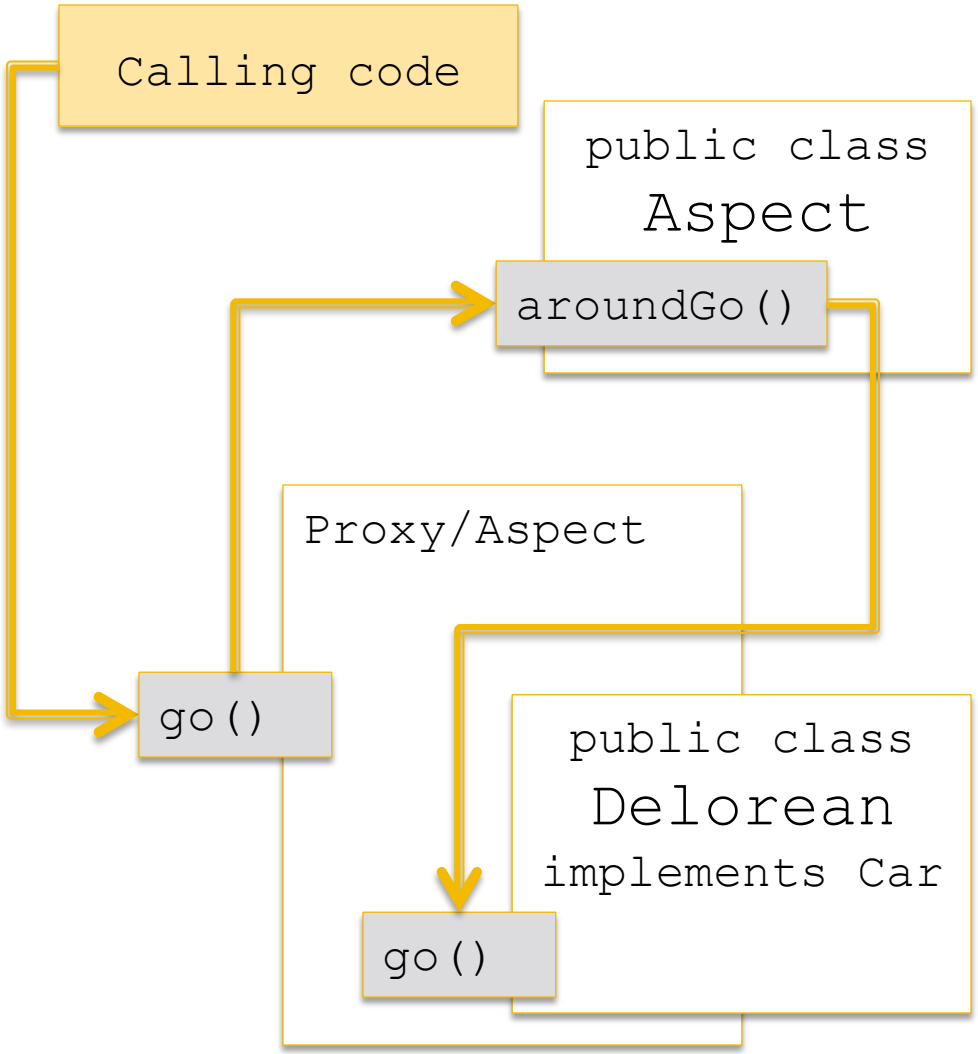


`BeanPostProcessor` used to wrap instances of `Car` with a Timer if they are annotated with `@BackToFuture`.

Spring AOP

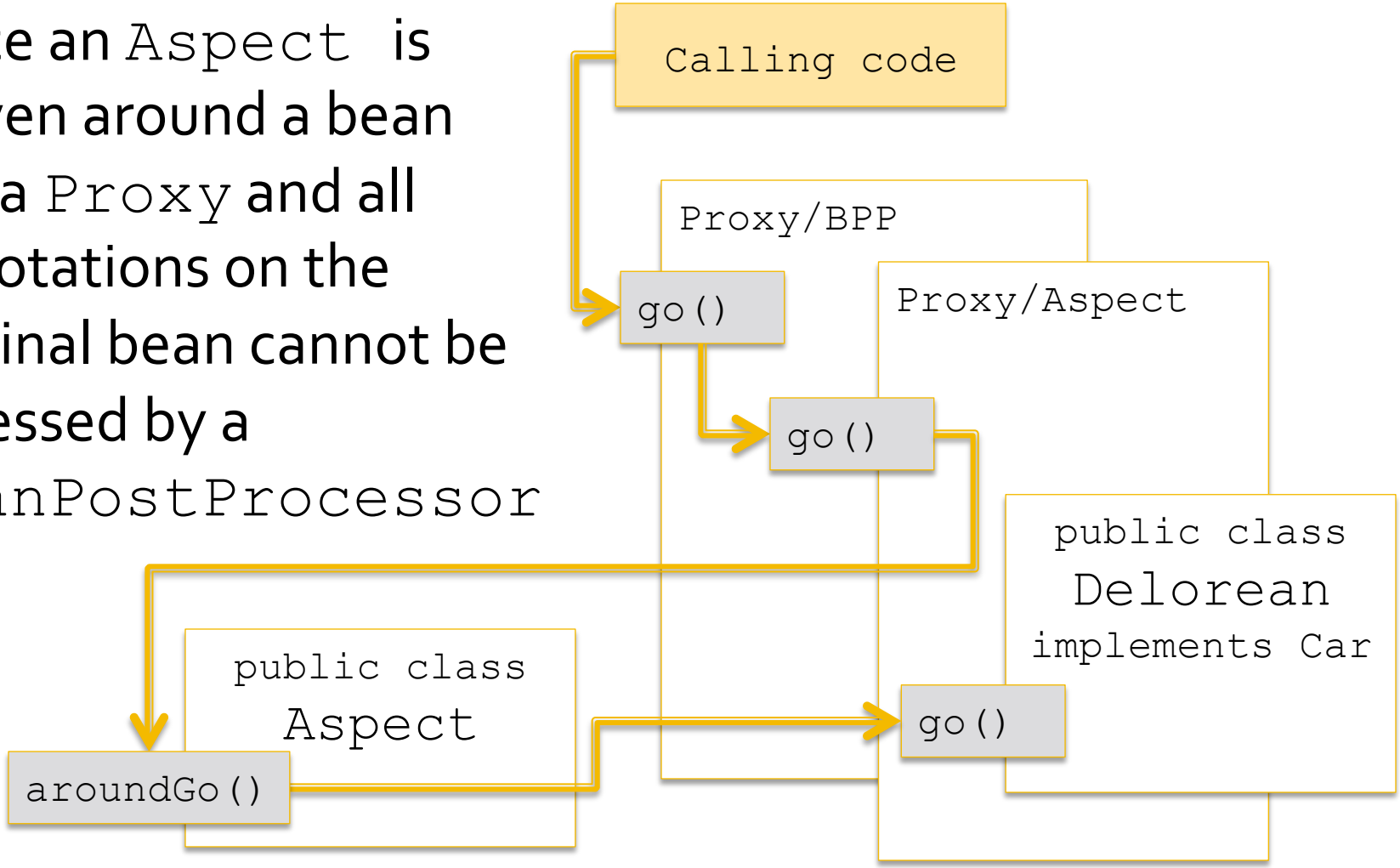


- Aspects are written using POJOs
  - Similar to writing a `BeanPostProcessor`
- Aspects are woven around managed beans using dynamic proxies
- Limited to method interception
  - Need to use AspectJ if need more than this



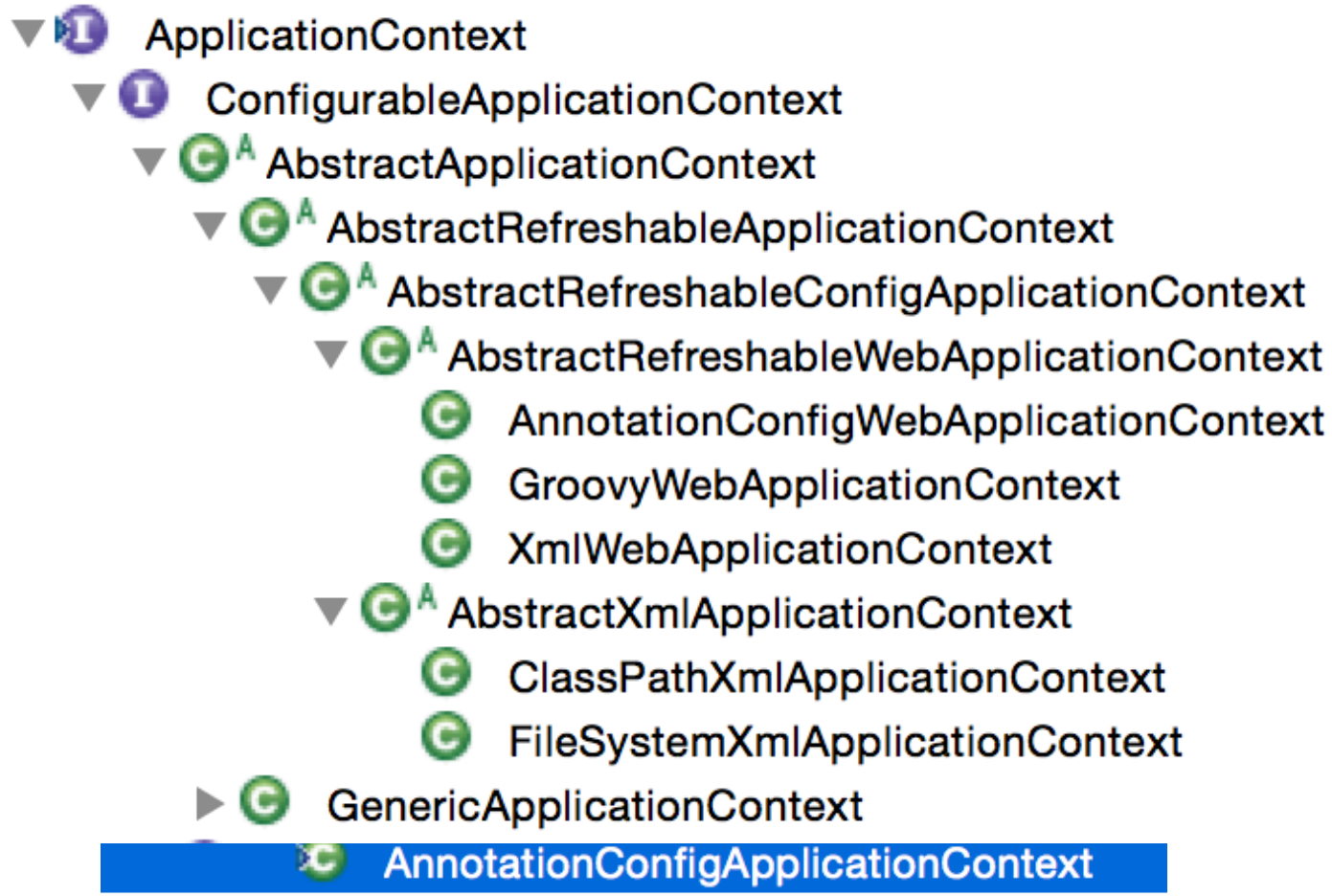
BPP and AOP

Once an `Aspect` is woven around a bean it is a `Proxy` and all annotations on the original bean cannot be accessed by a `BeanPostProcessor`

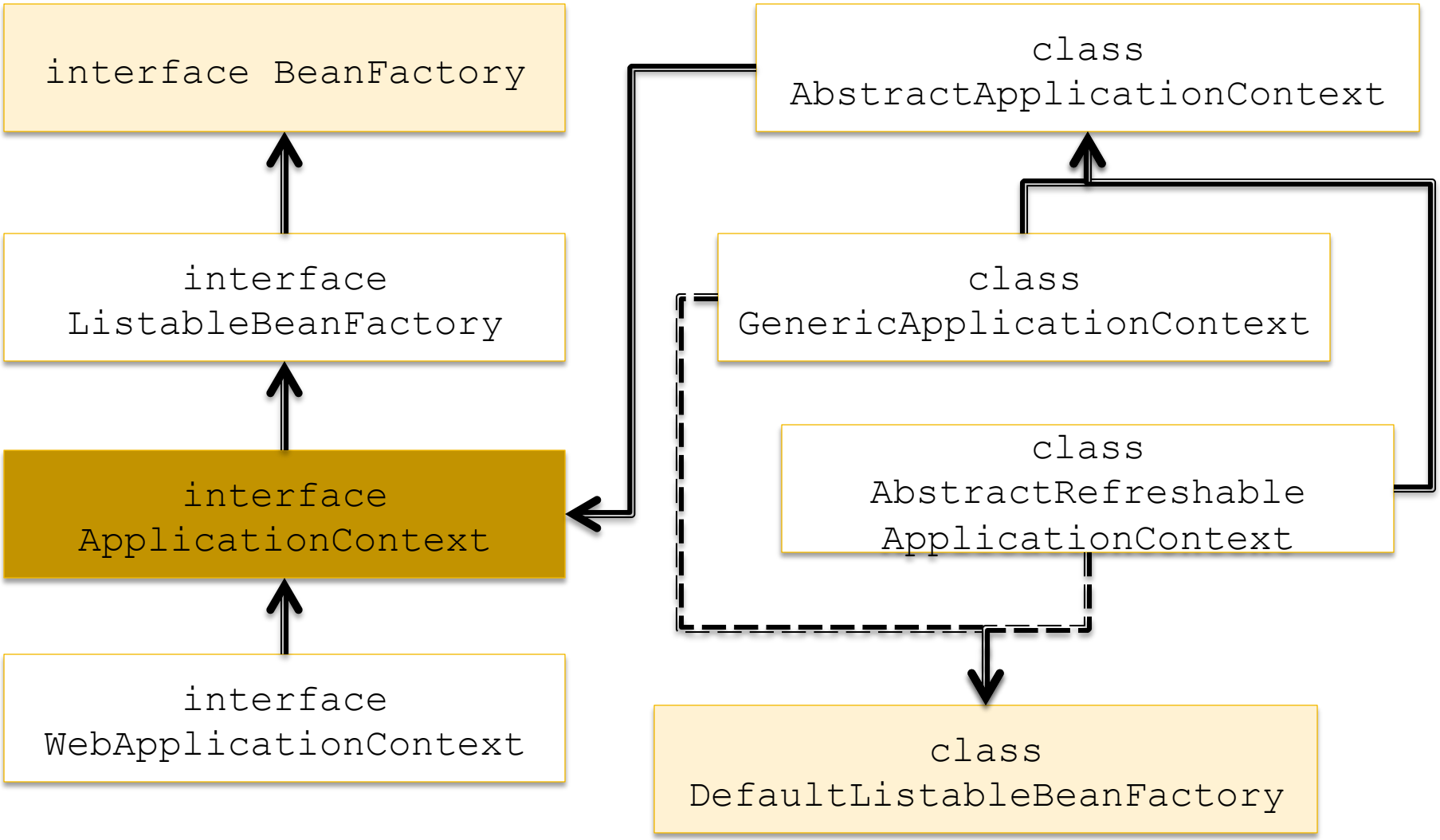


Q & A

ApplicationContext options



ApplicationContext uses DefaultListableBeanFactory



DefaultListableBeanFactory