The Great Springhoff's masterpiece

No one cares about the man in the box, the man who disappears...

Case description

It's been crowded the last few days in the city. We had a summer festival opening and it was attracting everyone's attention. Couples in love, families with children, tourists who wandered in by chance - it seemed that everyone could find something interesting to do

I came specially to see The Great Springhoff. Since childhood I love everything connected with riddles and secrets, and the famous magician's performance was something I just couldn't miss

Posters were promising something intriguing: the magician was going to get into a box, vanish from it in front of all watchers and then appear back again

I was wondering if I'll be able to understand, how he is doing his trick

Performance description on the poster

```
public static void main(String[] args) {
    System.out.println("Ladies and Gents, the main event of this evening is about to happen");

    // magician enters the box
    System.out.println("1st: Let's verify that The Great Springhoff is in the box");
    ConfigurableApplicationContext context = SpringApplication.run(Main.class, args);

    // dramatic pause
    System.out.println("2nd: Now, after just few seconds, we'll see The Great Springhoff is no more here");
    Watchable stage = context.getBean(Watchable.class);
    stage.watchVeryCarefully();

    // dramatic pause
    System.out.println("3rd: And now, just after few seconds again, let's open the box for the last time");
    context.close();

    // ovations
    System.out.println("That's all folks!");
}
```

So, the trick was going to happen this way: the magician will enter the box, then he was going to disappear, and then appear again

We'll be able to see what's inside the box 3 times

Stage description

When I entered the hall where the performance was going to happen, I immediately looked on the stage to see how everything was organized

There was a box and only the festival workers had access to work with it

In other hand, the stage itself was completely watchable from all sides and all visitors could look on what's happening

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

```java
public interface Watchable {
    void watchVeryCarefully();
}
```

```java
@Component
@ToString
public class Box {
    private final String magician = "Springhoff";
}
```

Attention on the performance

So, I decided to keep a high attention to all openings of the box

Whatever will happen, it should be somehow related to these openings (some hidden mechanism, reflective mirrors or something similar)

Right after I see anything suspicious, I'll understand how the trick is done

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

```
public @interface Attention {
}
```

```
@Aspect
@Component
public class AttentionAspect {
    @After("@annotation(detectivecases.case3.Attention)")
    public void sawSomethingSuspicious(){
        System.out.println("Now I understand how it works!");
    }
}
```

My expectations

```java
public static void main(String[] args) {
    System.out.println("Ladies and Gents, the main event of this evening is about to happen");
    // magician enters the box
    System.out.println("1st: Let's verify that The Great Springhoff is in the box");
    ConfigurableApplicationContext context = SpringApplication.run(Main.class, args);
    // dramatic pause
    System.out.println("2nd: Now, after just few seconds, we'll see The Great Springhoff is no more here");
    Watchable stage = context.getBean(Watchable.class);
    stage.watchVeryCarefully();
    // dramatic pause
    System.out.println("3rd: And now, just after few seconds again, let's open the box for the last time");
    context.close();
    // ovations
    System.out.println("That's all folks!");
}
```

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

So, there will be 3 times to look inside the box:

1. Right after Springhoff gets into it (**@PostConstruct**)
2. In the middle of the trick (by calling **watchVeryCarefully** method)
3. In the very end to see the magician appearing back (**@PreDestroy**)
4. And each one opening will have my full attention

The show starts...

I was pretty sure The Great Springhoff couldn't do something that I wouldn't see. When the trick started, I starred my eyes on the stage:

Ladies and Gents, the main event of this evening is about to happen

1st: Let's see that The Great Springhoff is already in the box
Looking in the box: Box(magician=Springhoff)

2nd: Now, after just few seconds, let's open the box again
Looking in the box: null

3rd: And now, just after few seconds again, let's open the box for the last time
Looking in the box: Box(magician=Springhoff)

That's all folks!

was expecting to see "Now I understand how it works!"

I was really surprised!

1st – the magician vanished indeed, and I didn't get a bit of an idea how he did it

2nd and more important – the box was opened 3 times, but I didn't see anything at all

I was tricked, but how did it happen?

How did it happen?

You can find the project on:
https://github.com/WieRuindl/detective-cases/tree/master/case3

It contains all the code described on the previous slides

You're very welcome to take the project and try to figure out the case yourself ☺

There will be a **lunch talk on 16 May at 14:00 BG time** where I will explain step by step what and how exactly happened

Explanations. Part I
Design Patterns. Proxy

Explanations. Part I
Design Patterns. Proxy

Imagine a boy named Jonny, who have a cool red toy car, and Jonny absolutely loves playing with it

```java
public class ToyCar {
    public void drive() {
        System.out.println("Zoom zoom! The car is speeding ahead!");
    }
}
```

```java
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        toyCar.drive();
    }
}
```

```java
public static void main(String[] args) {
    Jonny jonny = new Jonny();
    jonny.play();
}
```

```
Zoom zoom! The car is speeding ahead!
```

Explanations. Part I
Design Patterns. Proxy

Imagine a boy named Jonny, who have a cool red toy car, and Jonny absolutely loves playing with it

But Jonny's mother doesn't allow Jonny only to play, she requires him first to do his homework, and clean the room after all the games

```
public class ToyCar {
    public void drive() {
        System.out.println("Zoom zoom! The car is speeding ahead!");
    }
}
```

```
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        doHomework();
        toyCar.drive();
        cleanRoom();
    }
    private void doHomework() {
        System.out.println("Doing homework...");
    }
    private void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

Explanations. Part I
Design Patterns. Proxy

Imagine a boy named Jonny, who have a cool red toy car, and Jonny absolutely loves playing with it

But Jonny's allow Jonny only to play im first to do his homework oom after all the games


```java
public class ToyCar {
    public void drive() {
        System.out.println("Zoom zoom! The car is speeding ahead!");
    }
}
```

```java
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        doHomework(); // Jonny doesn't want to do this
        toyCar.drive();
        cleanRoom(); // and this as well
    }
    private void doHomework() {
        System.out.println("Doing homework...");
    }
    private void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

Explanations. Part I
Design Patterns. Proxy

This is where the proxy pattern comes to help. How it works:

Explanations. Part I
Design Patterns. Proxy

This is where the proxy pattern comes to help. How it works:

1) First, Jonny needs to find a friend

```
@RequiredArgsConstructor
public class Friend {
    private final Jonny jonny;



}
```

Explanations. Part I
Design Patterns. Proxy

This is where the proxy pattern comes to help. How it works:

1) First, Jonny needs to find a friend
2) Then Jonny explains him how to do his homework and how to clean the room

```java
@RequiredArgsConstructor
public class Friend {
    private final Jonny jonny;
    public void helpJonny() {
        doHomework();
        jonny.play();
        cleanRoom();
    }
    private void doHomework() {
        System.out.println("Doing homework...");
    }
    private void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

Explanations. Part I
Design Patterns. Proxy

This is where the proxy pattern comes to help. How it works:

1) First, Jonny needs to find a friend
2) Then Jonny explains him how to do his homework and how to clean the room
3) ???

```java
@RequiredArgsConstructor
public class Friend {
    private final Jonny jonny;
    public void helpJonny() {
        doHomework();
        jonny.play();
        cleanRoom();
    }
    private void doHomework() {
        System.out.println("Doing homework...");
    }
    private void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

Explanations. Part I
Design Patterns. Proxy

This is where the proxy pattern comes to help. How it works:

1) First, Jonny needs to find a friend
2) Then Jonny explains him how to do his homework and how to clean the room
3) ???
4) PROFIT: Jonny doesn't need to deal with the boring part ever again – he could just delegate this to his friend and focus only on the fun part. Homework is done, room is clean – just with little help from his friend

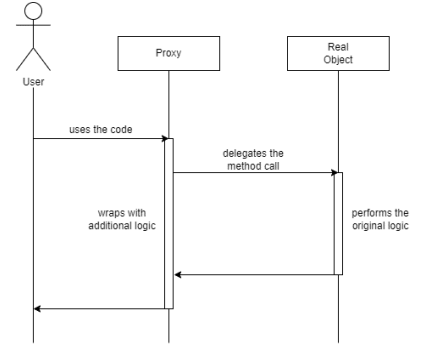
```
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        toyCar.drive(); // now Jonny only plays
    }
}
```

```
public static void main(String[] args) {
    Jonny jonny = new Jonny();
    Friend friend = new Friend(jonny);
    friend.helpJonny();
}
```

```
Doing homework...
Zoom zoom! The car is speeding ahead!
Cleaning room...
```

Explanations. Part I
Design Patterns. Proxy

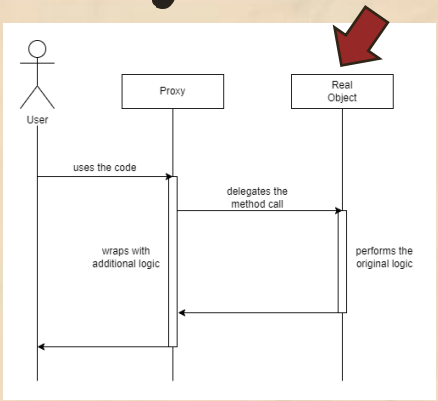
So, what's the point of proxy pattern:



Explanations. Part I
Design Patterns. Proxy

So, what's the point of proxy pattern:

1) we have the original object responsible only for the things we want him to be

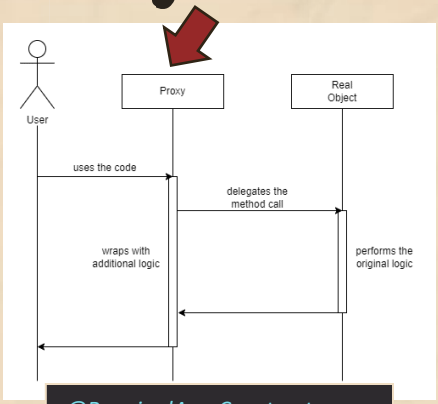



```
public class Jonny {
   private final ToyCar toyCar = new ToyCar();
   public void play() {
      toyCar.drive();
   }
}
```

Explanations. Part I
Design Patterns. Proxy

So, what's the point of proxy pattern:

1) we have the original object responsible only for the things we want him to be
2) we wrap it to a proxy object that could have any additional logic (caching/authorization/etc.)

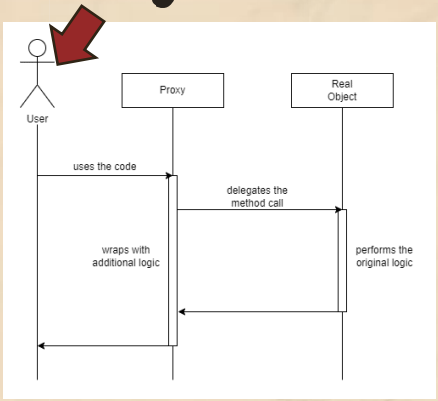



```
@RequiredArgsConstructor
public class Friend {
   private final Jonny jonny;
   public void helpJonny() {
      doHomework();
      jonny.play();
      cleanRoom();
   }
   // ...
}
```

Explanations. Part I
Design Patterns. Proxy

So, what's the point of proxy pattern:

1) we have the original object responsible only for the things we want him to be
2) we wrap it to a proxy object that could have any additional logic (caching/authorization/etc.)
3) we use the proxy object and have all of the functionality



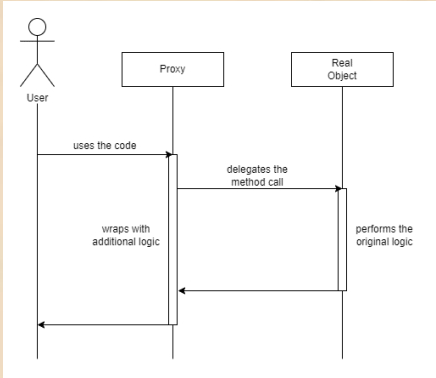

```
public static void main(String[] args) {
    Jonny jonny = new Jonny();
    Friend friend = new Friend(jonny);
    friend.helpJonny();
}
```

```
Doing homework...
Zoom zoom! The car is speeding ahead!
Cleaning room...
```

Explanations. Part I
Design Patterns. Proxy

So, what's the point of proxy pattern:

1) we have the original object responsible only for the things we want him to be
2) we wrap it to a proxy object that could have any additional logic (caching/authorization/etc.)
3) we use the proxy object and have all of the functionality
4) ???



Explanations. Part I
Design Patterns. Proxy

So, what's the point of proxy pattern:

1) we have the original object responsible only for the things we want him to be
2) we wrap it to a proxy object that could have any additional logic (caching/authorization/etc.)
3) we use the proxy object and have all of the functionality
4) ???
5) PROFIT: single responsibility, clean code and other good concepts

Explanations. Part II
Inversion of Control

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Explanations. Part II
Inversion of Control

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

```
public static void main(String[] args) {
    Jonny jonny = new Jonny();
    Friend friend = new Friend(jonny);
    friend.helpJonny();
}
```

Yet one thing remains not perfect

Explanations. Part II
Inversion of Control

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Yet one thing remains not perfect

Jonny needs to asks his friend for help each time he wants to play, otherwise or he'll have to do the boring things himself

```
public static void main(String[] args) {
    Jonny jonny = new Jonny();
    Friend friend = new Friend(jonny); // we should know about the friend

    friend.helpJonny(); // ask him for help
    friend.helpJonny(); // and again
    friend.helpJonny(); // and again..
}
```

Explanations. Part II
Inversion of Control

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Yet one thing re fect

Jonny needs iend for help each time he otherwise or he'll have to things himself

That's not very fun!

```
public static void main(String[] args) {
    Jonny jonny = new Jonny();
    Friend friend = new Friend(jonny); // we should know about the friend

    friend.helpJonny(); // ask him for help
    friend.helpJonny(); // and again
    friend.helpJonny(); // and again..
}
```

Explanations. Part II
Inversion of Control

Let's change this

Explanations. Part II
Inversion of Control

Let's change this

1) we create a proxy object

```
public class Friend {
    @SneakyThrows
    public static <T> T provideConstantHelp(Class<T> clazz) {
        ProxyFactory factory = new ProxyFactory();
        factory.setSuperclass(clazz);
        T proxy = (T) factory.createClass().getDeclaredConstructor().newInstance();
        return proxy;
    }
    private static void doHomework() {
        System.out.println("Doing homework...");
    }
    private static void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

Explanations. Part II
Inversion of Control

Let's change this

1) we create a proxy object
2) we change its methods invocations by adding the additional logic

```
public class Friend {
    @SneakyThrows
    public static <T> T provideConstantHelp(Class<T> clazz) {
        ProxyFactory factory = new ProxyFactory();
        factory.setSuperclass(clazz);
        T proxy = (T) factory.createClass().getDeclaredConstructor().newInstance();
        ((Proxy) proxy).setHandler((self, thisMethod, proceed, args) -> {
            doHomework();
            proceed.invoke(self, args);
            cleanRoom();
            return self;
        });
        return proxy;
    }
    private static void doHomework() {
        System.out.println("Doing homework...");
    }
    private static void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

Explanations. Part II
Inversion of Control

Let's change this

1) we create a proxy object
2) we change its methods invocations by adding the additional logic
3) we return proxy of the requested object with the *same type*

```
public class Friend {
  @SneakyThrows
  public static <T> T provideConstantHelp(Class<T> clazz) {
    ProxyFactory factory = new ProxyFactory();
    factory.setSuperclass(clazz);
    T proxy = (T) factory.createClass().getDeclaredConstructor().newInstance();
    ((Proxy) proxy).setHandler((self, thisMethod, proceed, args) -> {
      doHomework();
      proceed.invoke(self, args);
      cleanRoom();
      return self;
    });
    return proxy;
  }
  private static void doHomework() {
    System.out.println("Doing homework...");
  }
  private static void cleanRoom() {
    System.out.println("Cleaning room...");
  }
}
```

Explanations. Part II
Inversion of Control

Let's change this

1) we create a proxy object
2) we change its methods invocations by adding the additional logic
3) we return proxy of the requested object with the same type
4) ???

```
public class Friend {
  @SneakyThrows
  public static <T> T provideConstantHelp(Class<T> clazz) {
    ProxyFactory factory = new ProxyFactory();
    factory.setSuperclass(clazz);
    T proxy = (T) factory.createClass().getDeclaredConstructor().newInstance();
    ((Proxy) proxy).setHandler((self, thisMethod, proceed, args) -> {
      doHomework();
      proceed.invoke(self, args);
      cleanRoom();
      return self;
    });
    return proxy;
  }
  private static void doHomework() {
    System.out.println("Doing homework...");
  }
  private static void cleanRoom() {
    System.out.println("Cleaning room...");
  }
}
```

Explanations. Part II
Inversion of Control

Let's change this

1) we create a proxy object
2) we change its methods invocations by adding the additional logic
3) we return proxy of the requested object with the same type
4) ???
5) PROFIT: Jonny doesn't need to ask for his friend's help ever again. He already is aware of is duties and will perform them all by himself in the moment Jonny decides to play

```
public static void main(String[] args) {
    Jonny jonny = Friend.provideConstantHelp(Jonny.class);

    jonny.play(); // perfect
    jonny.play();
    jonny.play();
}
```

```
Doing homework...
Zoom zoom! The car is speeding ahead!
Cleaning room...
Doing homework...
Zoom zoom! The car is speeding ahead!
Cleaning room...
Doing homework...
Zoom zoom! The car is speeding ahead!
Cleaning room...
```

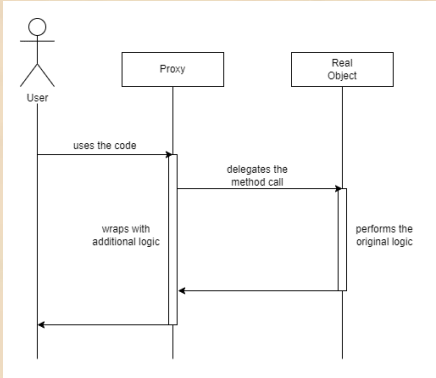
Explanations. Part II
Inversion of Control

So, what's the point of this modification:

Explanations. Part II
Inversion of Control

So, what's the point of this modification:

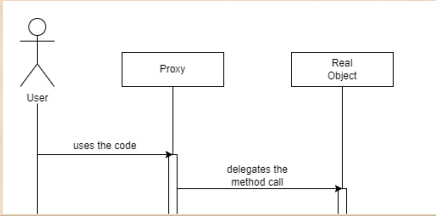
1) We still have all the cool functionality our proxy provides us



Explanations. Part II
Inversion of Control

So, what's the point of this modification:

1) We still have all the cool functionality our proxy provides us
2) But earlier we had to know about the proxy object, handle its creation and remembering using it instead of the original object

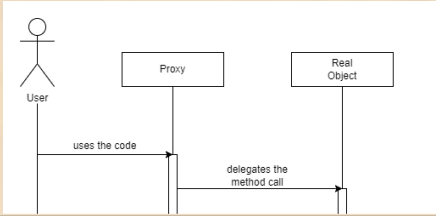



```
public static void main(String[] args) {
  Jonny jonny = new Jonny();
  Friend friend = new Friend(jonny); // we manually create proxy
  friend.helpJonny();
  // jonny.play(); // additional logic won't work
}
```

Explanations. Part II
Inversion of Control

So, what's the point of this modification:

1) We still have all the cool functionality our proxy provides us
2) But earlier we had to know about the proxy object, handle its creation and remembering using it instead of the original object
3) ???




```
public static void main(String[] args) {
  Jonny jonny = new Jonny();
  Friend friend = new Friend(jonny); // we manually create proxy
  friend.helpJonny();
  // jonny.play(); // additional logic won't work
}
```

Explanations. Part II
Inversion of Control

So, what's the point of this modification:

1) We still have all the cool functionality our proxy provides us
2) But earlier we had to know about the proxy object, handle its creation and remembering using it instead of the original object
3) ???
4) PROFIT: Now we don't care about this – everything happens automatically behind the scenes. We just use the original object (which already is a proxy) and everything works as we want


```
public static void main(String[] args) {
    Jonny jonny = Friend.provideConstantHelp(Jonny.class);
    jonny.play(); // using the code is simple and pleasant
}
```

Explanations. Part III
Annotations

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Explanations. Part III
Annotations

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Until one day he decides not to play with the car, but to go for a walk

```
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        toyCar.drive();
    }
    public void goForAWalk() {
        System.out.println("Such a sunny day!");
    }
}
```

Explanations. Part III
Annotations

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Until one day he decides not to play with the car, but to go for a walk

```
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        toyCar.drive();
    }
    public void goForAWalk() {
        System.out.println("Such a sunny day!");
    }
}
```

```
public static void main(String[] args) {
    Jonny jonny = Friend.provideConstantHelp(Jonny.class);
    jonny.goForAWalk();
}
```

Explanations. Part III
Annotations

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Until one day he decides not to play with the car, but to go for a walk

And unexpectedly his friend appeared to provide his helping assistance

```
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        toyCar.drive();
    }
    public void goForAWalk() {
        System.out.println("Such a sunny day!");
    }
}
```

```
public static void main(String[] args) {
    Jonny jonny = Friend.provideConstantHelp(Jonny.class);
    jonny.goForAWalk();
}
```

```
Doing homework...
Such a sunny day!
Cleaning room...
```

Explanations. Part III
Annotations

Jonny is happy now. He could play with his cool red car whenever he wants, and all the boring staff is getting done by his friend

Until one day h to play with the car, but t

And unexpec appeared to provide his h nce

Well, this actually is funny, but not what we want it to be ☹

```java
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    public void play() {
        toyCar.drive();
    }
    public void goForAWalk() {
        System.out.println("Such a sunny day!");
    }
}
```

```java
public static void main(String[] args) {
    Jonny jonny = Friend.provideConstantHelp(Jonny.class);
    jonny.goForAWalk();
}
```

```
Doing homework...
Such a sunny day!
Cleaning room...
```

Explanations. Part III
Annotations

This happens because of the creation
logic of our proxy processor

Explanations. Part III
Annotations

This happens because of the creation logic of our proxy processor

We take the requested class and wrap **all** its methods with additional logic

```
public class Friend {
    @SneakyThrows
    public static <T> T provideConstantHelp(Class<T> clazz) {
        ProxyFactory factory = new ProxyFactory();
        factory.setSuperclass(clazz);
        T proxy = (T) factory.createClass().getDeclaredConstructor().newInstance();
        ((Proxy) proxy).setHandler((self, thisMethod, proceed, args) -> {
            doHomework();
            proceed.invoke(self, args);
            cleanRoom();
            return self;
        });
        return proxy;
    }
    private static void doHomework() {
        System.out.println("Doing homework...");
    }
    private static void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```

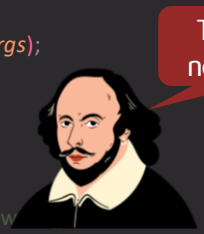
Explanations. Part III
Annotations

This happens because of the creation logic of our proxy processor

We take the requested class and wrap all its methods with additional logic

We need a way to say to this processor which methods should be wrapped with additional logic and which not

```
public class Friend {
    @SneakyThrows
    public static <T> T provideConstantHelp(Class<T> clazz) {
        ProxyFactory factory = new ProxyFactory();
        factory.setSuperclass(clazz);
        T proxy = (T) factory.createClass().getDeclaredConstructor().newInstance();
        ((Proxy) proxy).setHandler((self, thisMethod, proceed, args) -> {
            doHomework();
            proceed.invoke(self, args);
            cleanRoom();
            return self;
        });
        return proxy;
    }
    private static void doHomew
        System.out.println("Doing homework...");
    }
    private static void cleanRoom() {
        System.out.println("Cleaning room...");
    }
}
```



Explanations. Part III
Annotations

And this is where annotations come to help:

Explanations. Part III
Annotations

And this is where annotations come to help:

Annotations in Java are special markers that you can attach to different elements (classes, methods, etc.) of your code
They don't change how the code works themselves, but they provide additional information about the code to the compiler, runtime system, or other tools

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface FriendsHelp { }
```

```java
public class Jonny {
    private final ToyCar toyCar = new ToyCar();
    @FriendsHelp
    public void play() {
        toyCar.drive();
    }
    public void goingForAWalk() {
        System.out.println("Such a sunny day!");
    }
}
```

Explanations. Part III
Annotations

And this is where annotations come to help:

Annotations in Java are special markers that you can attach to different elements (classes, methods, etc.) of your code
They don't change how the code works themselves, but they provide additional information about the code to the compiler, runtime system, or other tools

Now we have complete annotation processor class that checks if annotation is presented and does all the magic behind it

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

```
public class Friend {
    @SneakyThrows
    public static <T> T provideConstantHelp(Class<T> clazz) {
        // ...
        ((Proxy) proxy).setHandler((self, thisMethod, proceed, args) -> {
            if (thisMethod.isAnnotationPresent(FriendsHelp.class)) {
                doHomework();
                proceed.invoke(self, args);
                cleanRoom();
                return self;
            } else {
                proceed.invoke(self, args);
                return self;
            }
        });
        return proxy;
    }
    // ...
}
```

Explanations. Part III
Annotations

So, we could create custom annotations and processors to work with them by adding logic or whatever we want them to be for

Explanations. Part III
Annotations

So, we could create custom annotations and processors to work with them by adding logic or whatever we want them to be for
Sooner or later, we'll find that we want these processors also to work automatically, finding all the classes they should modify themselves and we'll add reflection to scan all the classes in our code to track if corresponding annotation is presented

Explanations. Part III
Annotations

So, we could create custom annotations and processors to work with them by adding logic or whatever we want them to be for
Sooner or later, we'll find that we want these processors also to work automatically, finding all the classes they should modify themselves and we'll add reflection to scan all the classes in our code to track if corresponding annotation is presented
Then we'll find that scanning all the classes may be slow and invent annotations to mark which classes should be scanned

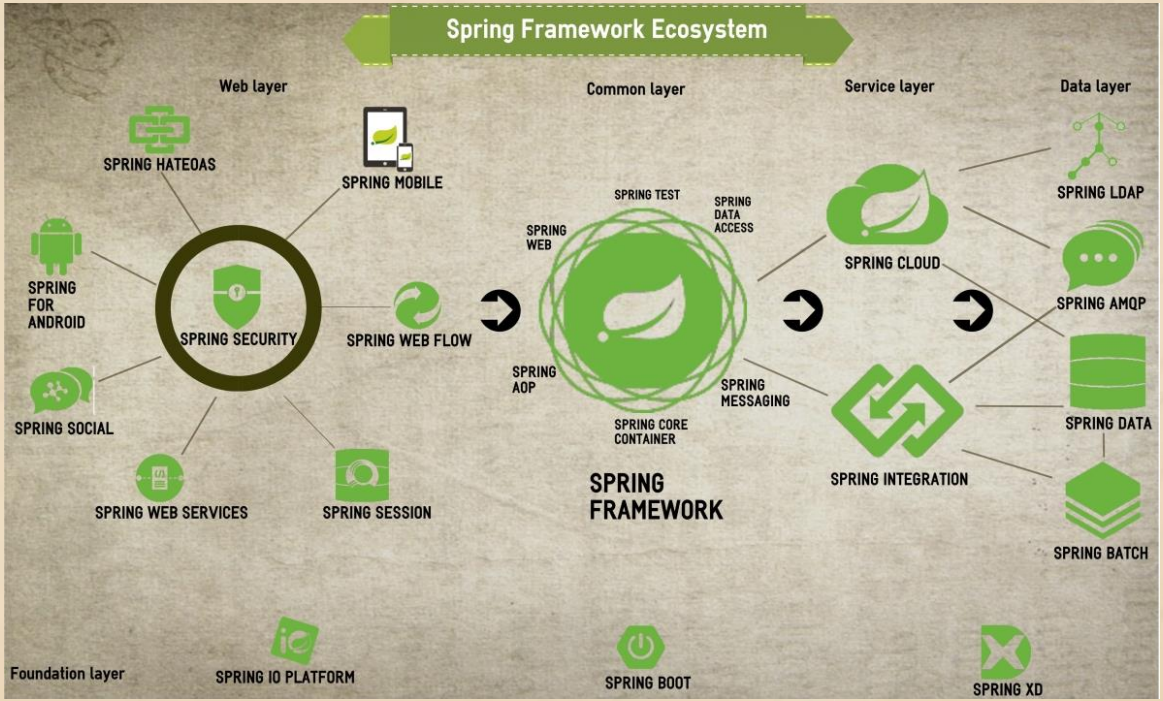
Explanations. Part III
Annotations

So, we could create custom annotations and processors to work with them by adding logic or whatever we want them to be for
Sooner or later, we'll find that we want these processors also to work automatically, finding all the classes they should modify themselves and we'll add reflection to scan all the classes in our code to track if corresponding annotation is presented
Then we'll find that scanning all the classes may be slow and invent annotations to mark which classes should be scanned

Eventually, after some such steps, we'll invent our own kind of

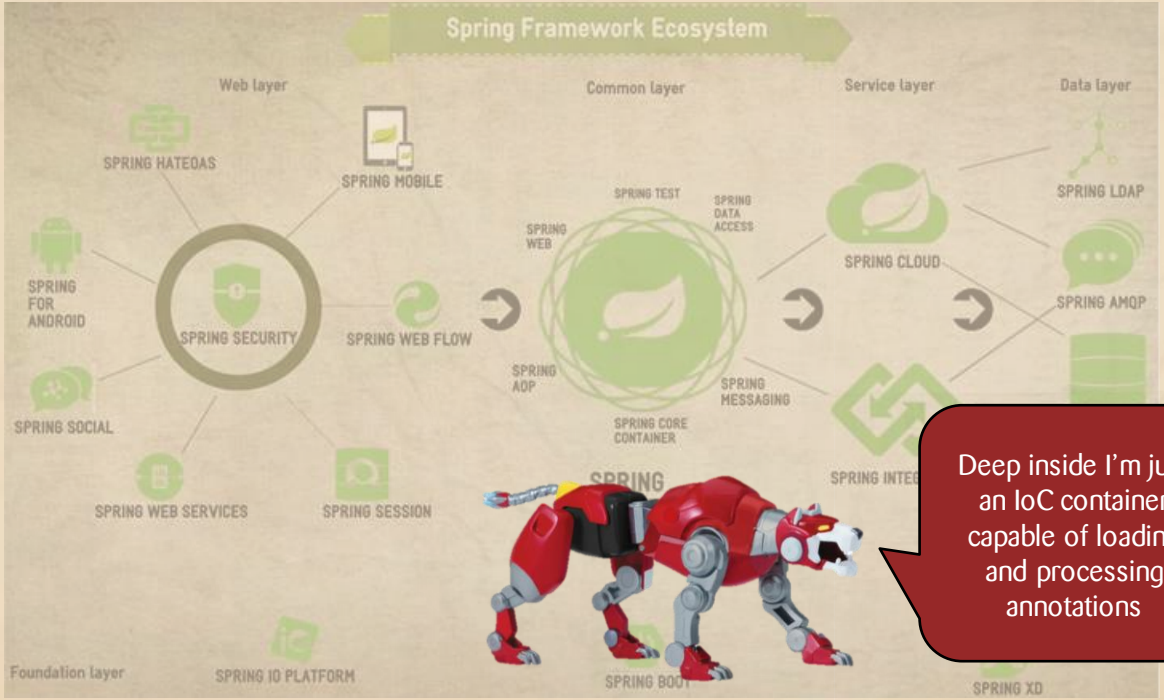
Explanations. Part IV
Spring Framework

Explanations. Part IV
Spring Framework



Explanations. Part IV
Spring Framework

Explanations. Part IV
Spring Framework



Explanations. Part IV
Spring beans lifecycle

Bean definitions are loaded

Loads bean definitions from xml file (don't use xml) | annotations with package scanning | @Configuration classes

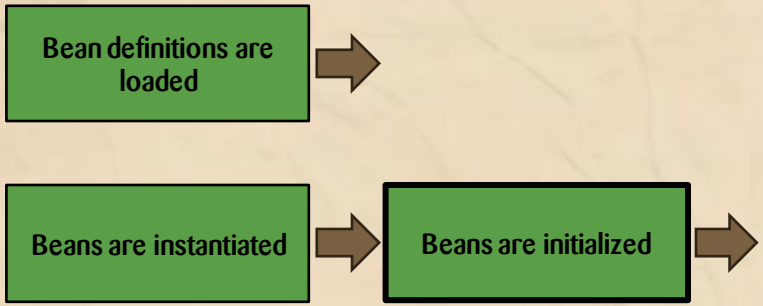
Explanations. Part IV
Spring beans lifecycle

Bean definitions are loaded

Beans are instantiated

Invokes constructors of each bean
If bean depends on another beans, builds them first
Injects dependencies via constructors, setters or @Autowired

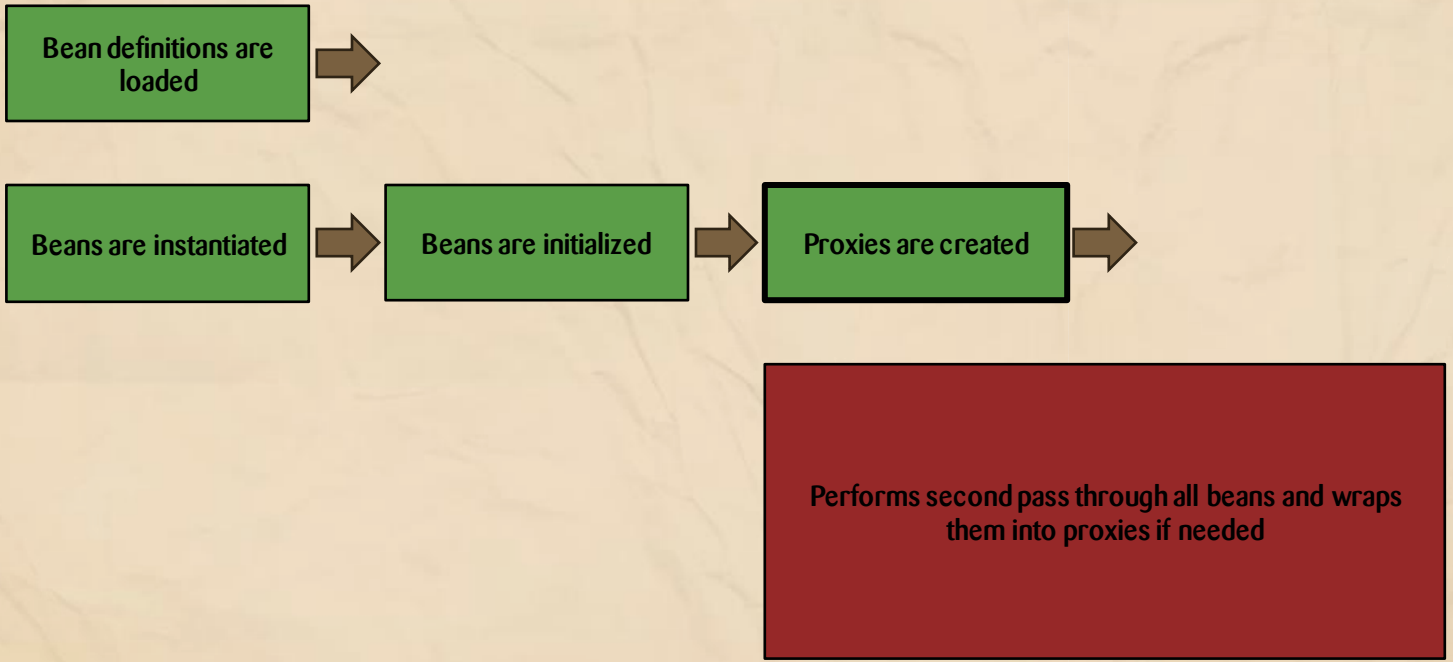
Explanations. Part IV
Spring beans lifecycle



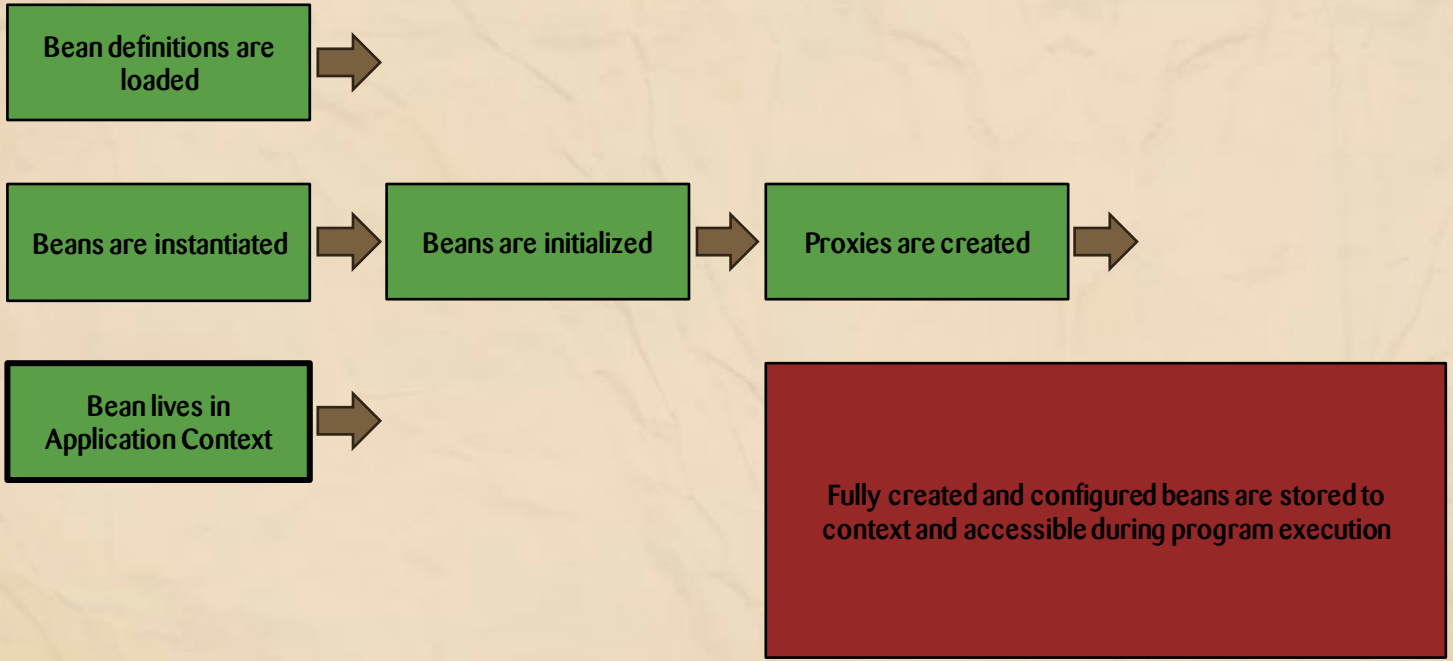
Triggers beans init methods:

- init-method of xml file (don't use xml)
- @PostConstruct of annotations-based config

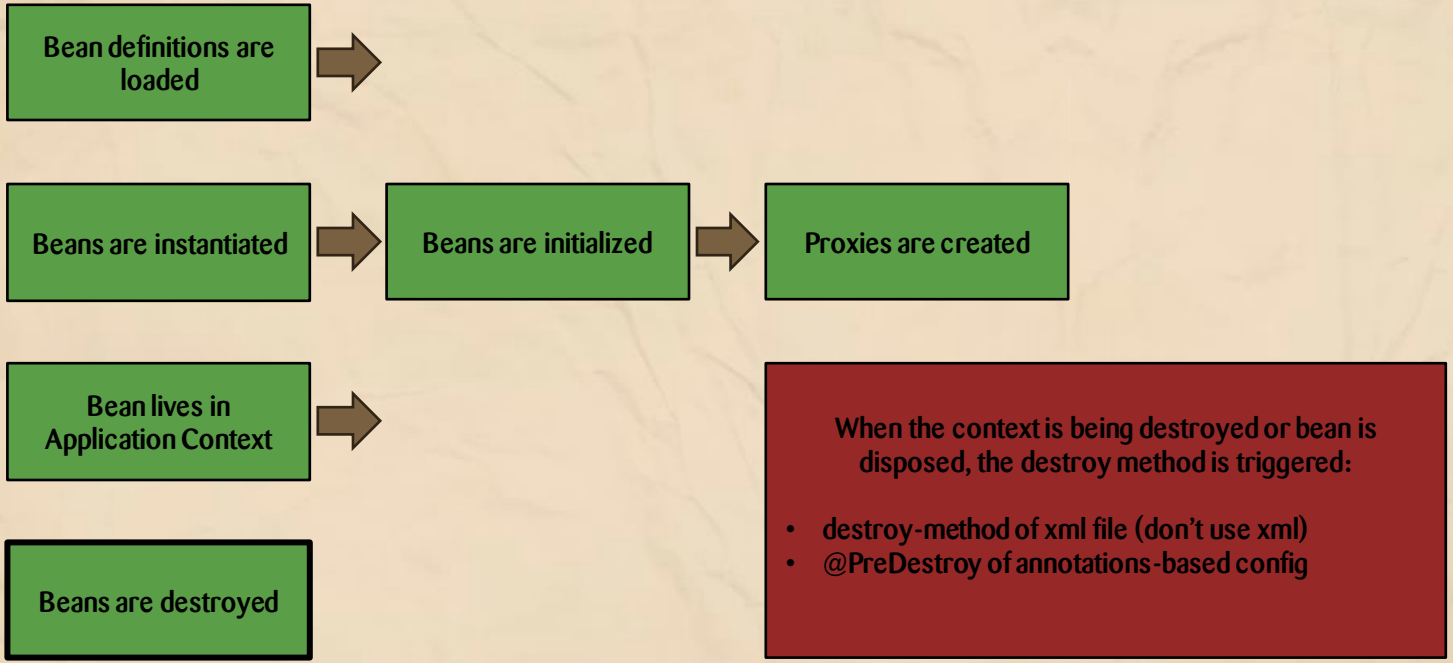
Explanations. Part IV
Spring beans lifecycle



Explanations. Part IV
Spring beans lifecycle



Explanations. Part IV
Spring beans lifecycle



Explanations. Part IV
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

Explanations. Part IV
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

```java
public @interface Attention {
}
```

```java
@Aspect
@Component
public class AttentionAspect {
    @After("@annotation(detectivecases.case3.Attention)")
    public void sawSomethingSuspicious(){
        System.out.println("Now I understand how it works!");
    }
}
```

Explanations. Part IV
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

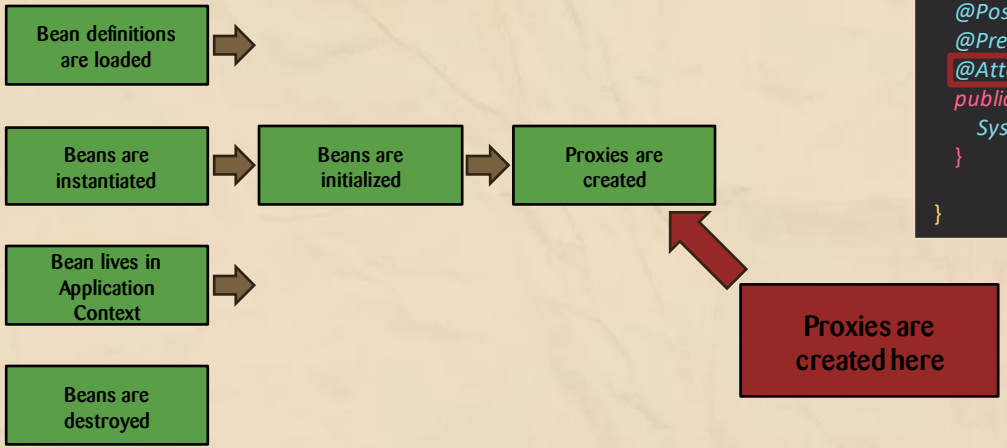

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

   private final Box box;

   @PostConstruct
   @PreDestroy
   @Attention
   public final void watchVeryCarefully() {
      System.out.println("Looking in the box: " + box);
   }

}
```

```
public @interface Attention {
}
```

```
@Aspect
@Component
public class AttentionAspect {
   @After("@annotation(detectivecases.case3.Attention)")
   public void sawSomethingSuspicious(){
      System.out.println("Now I understand how it works!");
   }
}
```

Explanations. Part IV
Revealing the mystery

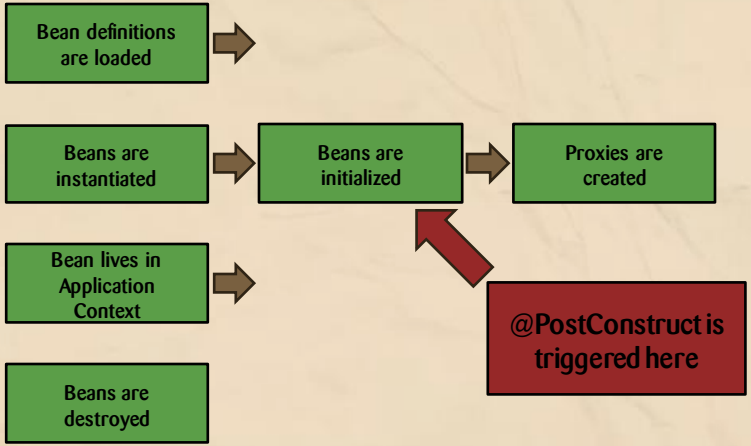
Now let's go back to The Great Sprighoff magic trick and see what is happening there:




```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

  private final Box box;

  @PostConstruct
  @PreDestroy
  @Attention
  public final void watchVeryCarefully() {
    System.out.println("Looking in the box: " + box);
  }

}
```

Explanations. Part IV
Revealing the mystery

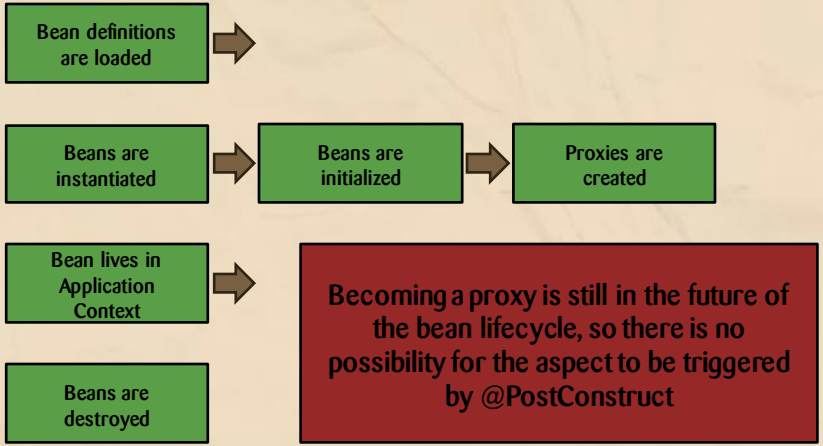
Now let's go back to The Great Sprighoff magic trick and see what is happening there:



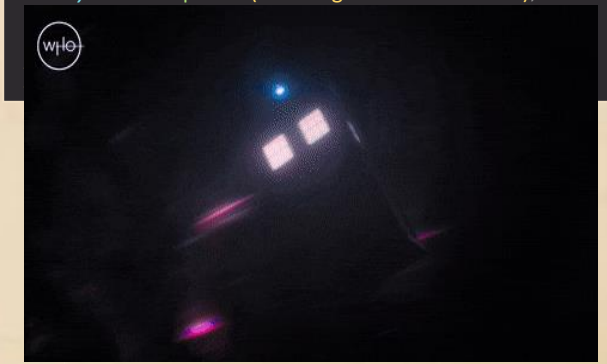

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

  private final Box box;

  @PostConstruct
  @PreDestroy
  @Attention
  public final void watchVeryCarefully() {
    System.out.println("Looking in the box: " + box);
  }

}
```

Explanations. Part IV
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

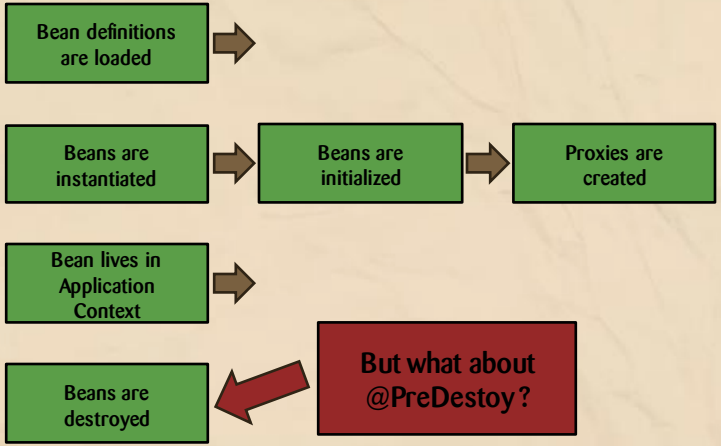



```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

  private final Box box;

  @PostConstruct
  @PreDestroy
  @Attention
  public final void watchVeryCarefully() {
    System.out.println("Looking in the box: " + box);
```



Explanations. Part IV
Revealing the mystery

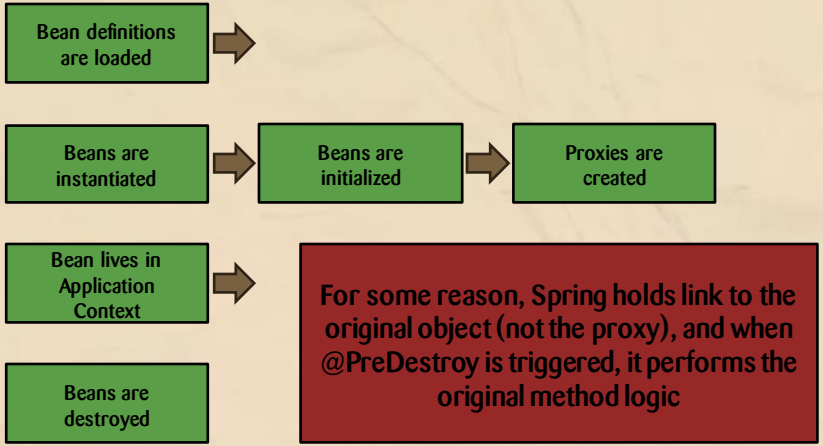
Now let's go back to The Great Sprighoff magic trick and see what is happening there:




```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

  private final Box box;

  @PostConstruct
  @PreDestroy
  @Attention
  public final void watchVeryCarefully() {
    System.out.println("Looking in the box: " + box);
```

Explanations. Part IV
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:




```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

  private final Box box;

  @PostConstruct
  @PreDestroy
  @Attention
  public final void watchVeryCarefully() {
    System.out.println("Looking in the box: " + box);
```

Explanations. Part IV
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

```
public static void main(String[] args) {
    // ...
    System.out.println("2nd: Now, after just few seconds, let's open the box again");
    Watchable stage = context.getBean(Watchable.class);
    stage.watchVeryCarefully();
    // ...
}
```

```
2nd: Now, after just few seconds, let's open the box again
Looking in the box: null
```

But why it didn't work here and why there's null ?

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

Explanations. Part V
Spring Proxies Creation

Well, the last step – how Spring manages to create proxies:

Explanations. Part V
Spring Proxies Creation

Well, the last step – how Spring manages to create proxies:

There are two possible mechanisms in Spring and the only thing the selection between them depends on is...

JDK Proxy

CGLIB Proxy

Explanations. Part V
Spring Proxies Creation

Well, the last step – how Spring manages to create proxies:

There are two possible mechanisms in Spring and the only thing the selection between them depends on is...

If the class implements an interface or not

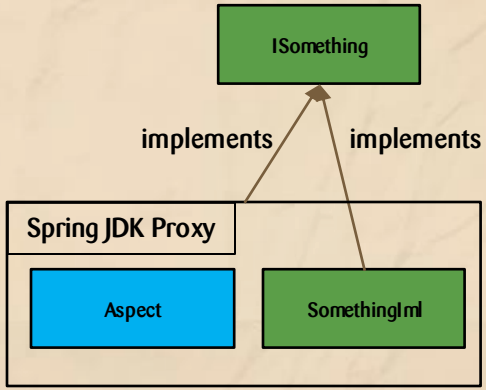
Explanations. Part V
Spring Proxies Creation

```java
public interface ISomething {
    void method();
}
```

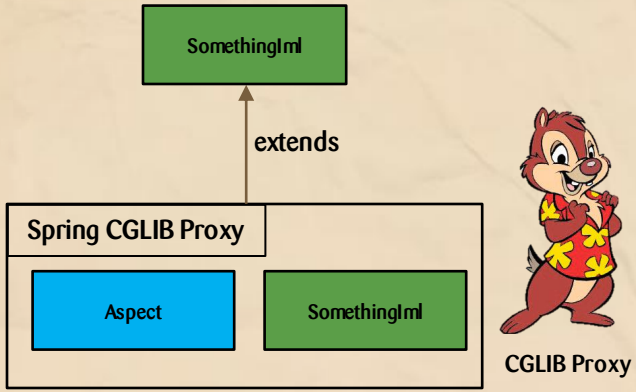
```java
public class SomethingImpl implements ISomething {
    @Override
    public void method() {
        // real object logic
    }
}
```

```java
public class SomethingJdkProxyImpl implements ISomething {
    ISomething realObject;
    @Override
    public void method() {
        // place where we can add new proxy logic
        realObject.method(); // the real method calling
        // place where we can add new proxy logic
    }
}
```

JDK Proxy



Explanations. Part V
Spring Proxies Creation



CGLIB Proxy

```java
public class SomethingImpl {
    public void method() {
        // real object logic
    }
}
```

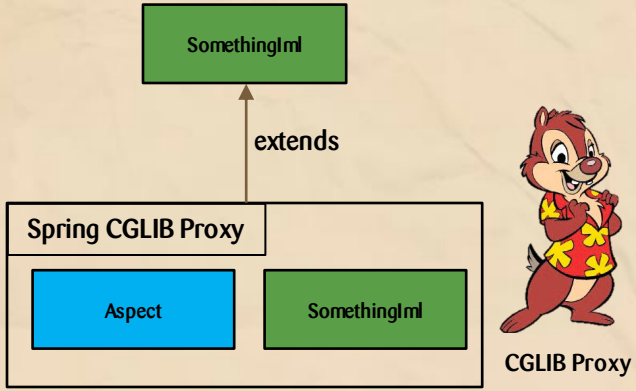
```java
public class SomethingCglibProxyImpl extends SomethingImpl {
    SomethingImpl realObject;
    @Override
    public void method() {
        // place where we can add logic
        realObject.method(); // the real method calling
        // place where we can add logic
    }
}
```

Explanations. Part V
Spring Proxies Creation

```
public class Stage$$SpringCGLIB$$0 extends Stage implements SpringProxy, Advised, Factory {
    // ...
    private static final Method CGLIB$watchVeryCarefully$0$Method;
    private static final MethodProxy CGLIB$watchVeryCarefully$0$Proxy;
    // ...
    CGLIB$watchVeryCarefully$0$Method = ReflectUtils.findMethods(
        new String[]{"watchVeryCarefully", "()V"},
        (var1 = Class.forName("detectivecases.case3.stage.Stage"))
        .getDeclaredMethods())[0];
    CGLIB$watchVeryCarefully$0$Proxy = MethodProxy.create(var1, var0, "()V", "watchVeryCarefully", "CGLIB$watchVeryCarefully$0");
    // ...
```

```
public final void watchVeryCarefully() {
    MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
    if (var10000 == null) {
        CGLIB$BIND_CALLBACKS(this);
        var10000 = this.CGLIB$CALLBACK_0;
    }
    if (var10000 != null) {
        var10000.intercept(this, CGLIB$watchVeryCarefully$0$Method, CGLIB$emptyArgs, CGLIB$watchVeryCarefully$0$Proxy);
    } else {
        super.watchVeryCarefully();
    }
}
```

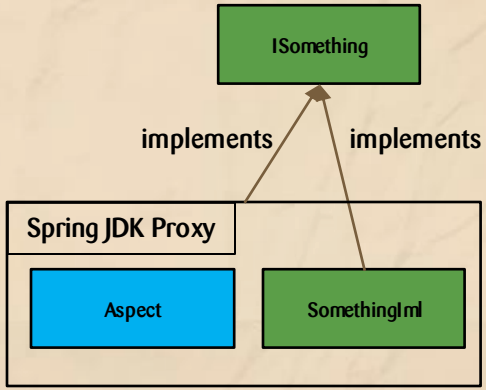
Explanations. Part V
Spring Proxies Creation



CGLIB Proxy

- can't proxy final classes
- can't override final or private methods

JDK Proxy



- can override only methods declared in the interface

Explanations. Part V
Spring Proxies Creation

Let's take a look on our code again:

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }
```

There is an Aspect here, so we need to create a proxy

Explanations. Part V
Spring Proxies Creation

Let's take a look on our code again:

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }
```

Explanations. Part V
Spring Proxies Creation

Let's take a look on our code again:

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }
```

Explanations. Part VI
Spring Boot

Explanations. Part VI
Spring Boot

Being a framework extension to Spring, Spring Boot makes working with Spring much easier and faster, providing developers with a set of utilities that automate Spring configuration

```
@SpringBootApplication
@RequiredArgsConstructor
public class Main {

    public static void main(String[] args) {
        // ...
    }
}
```

Explanations. Part VI
Spring Boot

BUT Spring Boot AOP is not the same as Spring AOP

```
@SpringBootApplication
@RequiredArgsConstructor
public class Main {

    public static void main(String[] args) {
        // ...
    }
}
```

```
@AutoConfiguration
@ConditionalOnProperty(prefix = "spring.aop", name = "auto", havingValue = "true", matchIfMissing = true)
public class AopAutoConfiguration {

    static class AspectJAutoProxyingConfiguration {
        @Configuration(proxyBeanMethods = false)
        @EnableAspectJAutoProxy(proxyTargetClass = false)
        @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "false")
        static class JdkDynamicAutoProxyConfiguration {}

        @Configuration(proxyBeanMethods = false)
        @EnableAspectJAutoProxy(proxyTargetClass = true)
        @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "true",
            matchIfMissing = true)
        static class CglibAutoProxyConfiguration {}
    }
```

Explanations. Part VI
Spring Boot

BUT Spring Boot AOP is not the same as Spring AOP

And the default mechanism is CGLIB proxies

```java
@SpringBootApplication
@RequiredArgsConstructor
public class Main {

    public static void main(String[] args) {
        // ...
    }
}
```

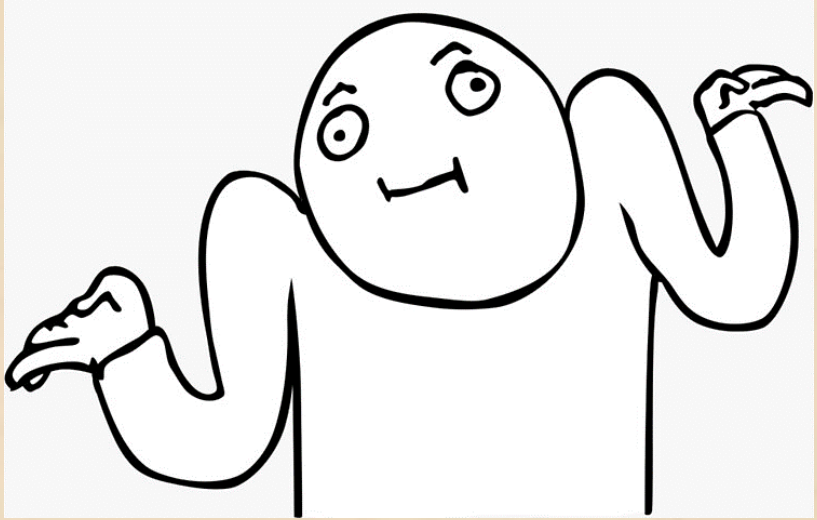
```java
@AutoConfiguration
@ConditionalOnProperty(prefix = "spring.aop", name = "auto", havingValue = "true", matchIfMissing = true)
public class AopAutoConfiguration {

    static class AspectJAutoProxyingConfiguration {
        @Configuration(proxyBeanMethods = false)
        @EnableAspectJAutoProxy(proxyTargetClass = false)
        @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "false")
        static class JdkDynamicAutoProxyConfiguration {}

        @Configuration(proxyBeanMethods = false)
        @EnableAspectJAutoProxy(proxyTargetClass = true)
        @ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class", havingValue = "true",
            matchIfMissing = true)
        static class CglibAutoProxyConfiguration {}
    }
```

Explanations. Part VI
Spring Boot

Why it's done this way?

Explanations. Part VI
Spring Boot

Why it's done this way?



Explanations. Part VI
Spring Boot

Assumption

Many developers don't like to code against interfaces. The only reasons to introduce one for them are either:

1) they have multiple implementations of this interface

or

2) they're developing some API which is going to be shared

So, they either don't introduce interfaces at all, or even if they do, in the dependent classes they could declare the bean with the type of the concrete implementation class, not the interface type

Explanations. Part VI
Spring Boot

If the default logic was to use JDK proxies, this code just wouldn't work:

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

```java
public static void main(String[] args) {
    // ...
    Stage stage = context.getBean(Stage.class);
    // ...
}
```

```java
@Autowired
private final Stage stage;
```

Explanations. Part VI
Spring Boot

If the default logic was to use JDK proxies, this code just wouldn't work:

```java
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

```java
public static void main(String[] args) {
    // ...
    Stage stage = context.getBean(Stage.class);
    // ...
}
```

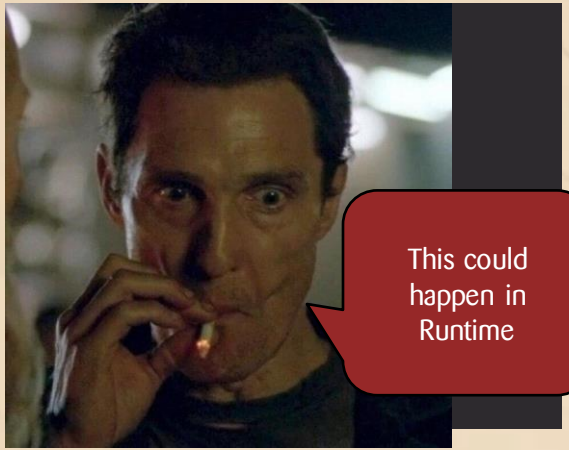
```java
@Autowired
private final Stage stage;
```

It's not a **Stage** class object anymore

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type 'detectivecases.case3.stage.Stage' available
...
```

Explanations. Part VI
Spring Boot

If the default logic was to use JDK proxies, this code just wouldn't work:




```
public static void main(String[] args) {
    // ...
    Stage stage = context.getBean(Stage.class);
    // ...
}
```

```
@Autowired
private final Stage stage;
```

It's jdk.proxy2.$Proxy41

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type 'detectivecases.case3.stage.Stage' available
...
```

Personal opinion

I'll make a bit "holy war" point, but

Personal opinion

I'll make a bit "holy war" point, but

It's ALWAYS better to introduce an interface and work against it

Personal opinion

I'll make a bit "holy war" point, but

It's ALWAYS better to introduce an interface and work against it

The only disadvantage of such a decision would be a bit more complicated set-up: defining interfaces and implementing them requires additional effort compared to using concrete classes directly, especially for simple cases

The benefits are far more:

- Modularity: separating the contract from the implementation
- Flexibility: modify implementations without changing the client code
- Testability: easier to mock dependencies during unit testing

Explanations. Part VII
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

```
public static void main(String[] args) {
    // ...
    System.out.println("2nd: Now, after just few seconds, let's open the box again");
    Watchable stage = context.getBean(Watchable.class);
    stage.watchVeryCarefully();
    // ...
}
```

```
2nd: Now, after just few seconds, let's open the box again
Looking in the box: null
```

We stopped on the missing aspect and null value

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

Explanations. Part VII
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

Both issues are related to the final keyword

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

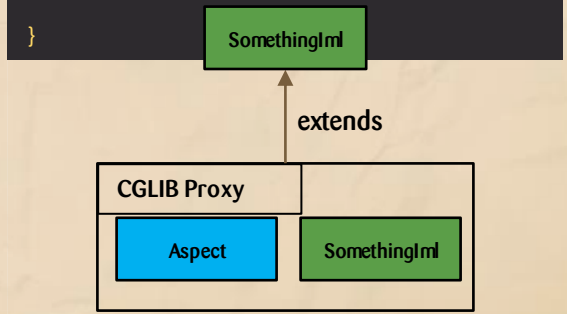
Explanations. Part VII
Revealing the mystery

Now let's go back to The Great Sprighoff magic trick and see what is happening there:

Both issues are related to the final keyword

Because of our proxy is inherited class, it could not override final methods. The code remains the same, so aspect logic could not be added – this is why there is no aspect message in the console

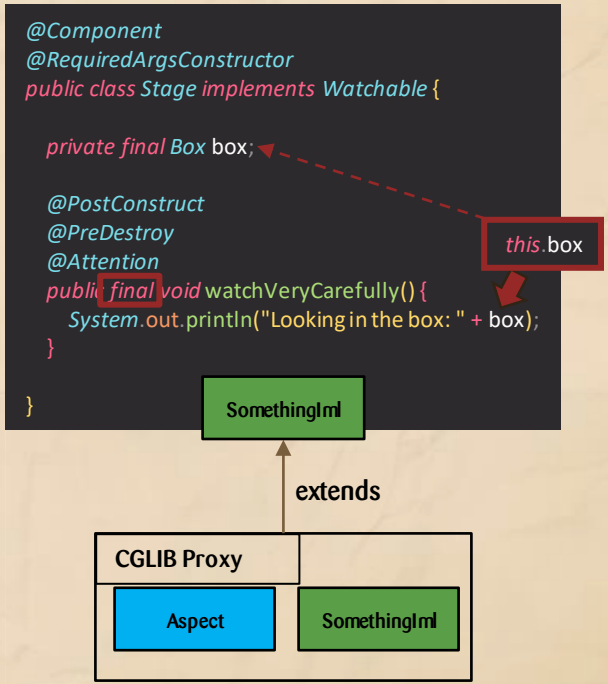
```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```



Explanations. Part VII
Revealing the mystery

But not only the missing message

Because proxy couldn't refer to the real object, it takes its own field in the method



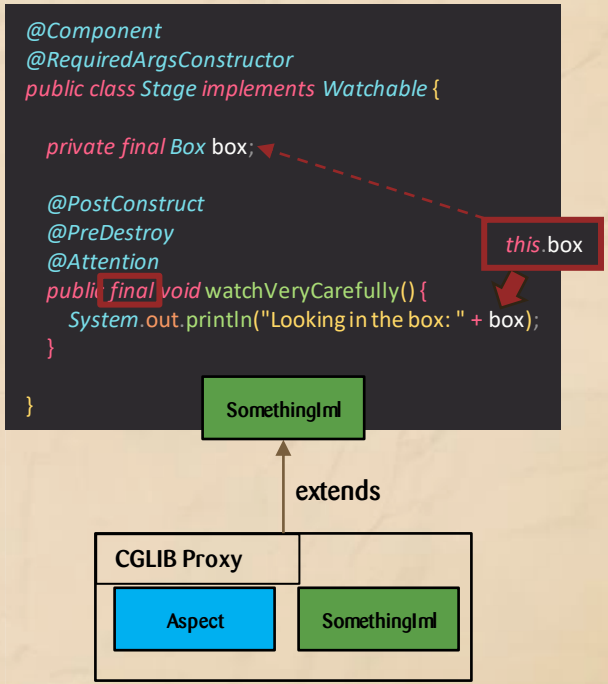
Explanations. Part VII
Revealing the mystery

But not only the missing message

Because proxy couldn't refer to the real object, it takes its own field in the method

Despite CGLIB copies all the parent class fields, it doesn't copy all the values as well:

- **primitive types** and **String** values will be presented
- **reference types** fields will contain **null**



Explanations. Part VII
Revealing the mystery

Let's summarize everything:

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box:
" + box);
    }

}
```

Explanations. Part VII
Revealing the mystery

Let's summarize everything:

- During first classes scan and bean configuring, Spring performs **@PostConstruct** logic, but at that point there is still no proxy with the aspect logic – so there is no message

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box:
" + box);
    }

}
```

Explanations. Part VII
Revealing the mystery

Let's summarize everything:

- During first classes scan and bean configuring, Spring performs @PostConstruct logic, but at that point there is still no proxy with the aspect logic – so there is no message
- Because of the **@Attention** aspect, Spring creates CGLIB proxy to add the aspect logic

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box:
" + box);
    }

}
```

Explanations. Part VII
Revealing the mystery

Let's summarize everything:

- During first classes scan and bean configuring, Spring performs @PostConstruct logic, but at that point there is still no proxy with the aspect logic – so there is no message
- Because of the @Attention aspect, Spring creates CGLIB proxy to add the aspect logic
- The **final** keywords doesn't allow the proxy to override the original method, so there will be no aspect message

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box:
" + box);
    }

}
```

Explanations. Part VII
Revealing the mystery

Let's summarize everything:

- During first classes scan and bean configuring, Spring performs @PostConstruct logic, but at that point there is still no proxy with the aspect logic – so there is no message
- Because of the @Attention aspect, Spring creates CGLIB proxy to add the aspect logic
- The final keywords doesn't allow the proxy to override the original method, so there will be no aspect message
- Because of the method refers to the **box** field of the same class (i.e. proxy), not the real object one, box contains **null**

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " + box);
    }

}
```

Explanations. Part VII
Revealing the mystery

Let's summarize everything:

- During first classes scan and bean configuring, Spring performs @PostConstruct logic, but at that point there is still no proxy with the aspect logic – so there is no message
- Because of the @Attention aspect, Spring creates CGLIB proxy to add the aspect logic
- The final keywords doesn't allow the proxy to override the original method, so there will be no aspect message
- Because of the method refers to the box field of the same class (i.e. proxy), not the real object one, box contains null
- When application context closes and the **@PreDestroy** logic is called, the method of the original object (and not the proxy one) is called – so no message again

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box:
" + box);
    }

}
```

Hacking the trick

There is no way to make @PostConstruct and @PreDestroy work with the @Attention aspect. Yet there are some possible solutions at least not to let the magician vanish from the box:

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

   private final Box box;

   @PostConstruct
   @PreDestroy
   @Attention
   public final void watchVeryCarefully() {
      System.out.println("Looking in the box:
" + box);
   }

}
```

Hacking the trick

There is no way to make @PostConstruct and @PreDestroy work with the @Attention aspect. Yet there are some possible solutions at least not to let the magician vanish from the box:

1. Remove the final keyword. This will allow CGLIB to override this method, so aspect logic will be added, and calling to the box field will be referred to the original object
2. Add ***spring.aop.proxy-target-class=false*** to application.yml. This will change proxying mechanism from CGLIB to JDK proxies. Final methods are not problem for JDK proxies, it's only necessary the method to come from interface

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box:
" + box);
    }

}
```

```
2nd: Now, after just few seconds, let's open the box again
Looking in the box: Box(magician=Springhoff)
Now I understand how it works!
```

Hacking the trick

There is no way to make @PostConstruct and @PreDestroy work with the @Attention aspect. Yet there are some possible solutions at least not to let the magician vanish from the box:

1. Remove the final keyword. This will allow CGLIB to override this method, so aspect logic will be added, and calling to the box field will be referred to the original object.
2. Add *spring.aop.proxy-target-class=false* to application.yml. This will change proxying mechanism from CGLIB to JDK proxies. Final methods are not problem for JDK proxies, it's only necessary the method to come from interface

PROBLEM SOLVED

```
@Component
@RequiredArgsConstructor
public class Stage implements Watchable {

    private final Box box;

    @PostConstruct
    @PreDestroy
    @Attention
    public final void watchVeryCarefully() {
        System.out.println("Looking in the box: " +
box);
    }
}
```

2nd: Now, after just few seconds, let's open the box again
Looking in the box: Box(magician=Springhoff)
Now I understand how it works!

THANKS!

Do you have any questions?

youremail@freepik.com
+91 620 421 838
yourwebsite.com

Please keep this slide for attribution