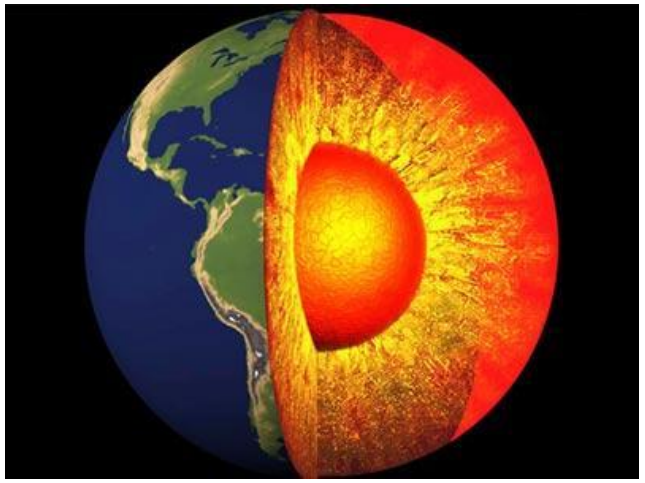


Spring - Core

Evgeny Borisov

evgeny@inwhite.pro

About myself

Developing Java since 2001

CEO of InWhite company

Partner of JFrog

Consulting

Lecturing

Writing courses

Writing code

Agenda

- Different Design Patterns
- Reflection – that’s cool!
- Main Spring concepts:
  - Inversion of control
  - Dependency injection
  - Spring Bean
  - BeanFactory
- Different context types

Agenda

- BeanPostProcessors
  - 4 levels of understanding of BeanPostProcessors
  - Write your own BeanPostProcessora
  - Write advanced BeanPostProcessora
- BeanFactoryPostProcessors
- ApplicationListener

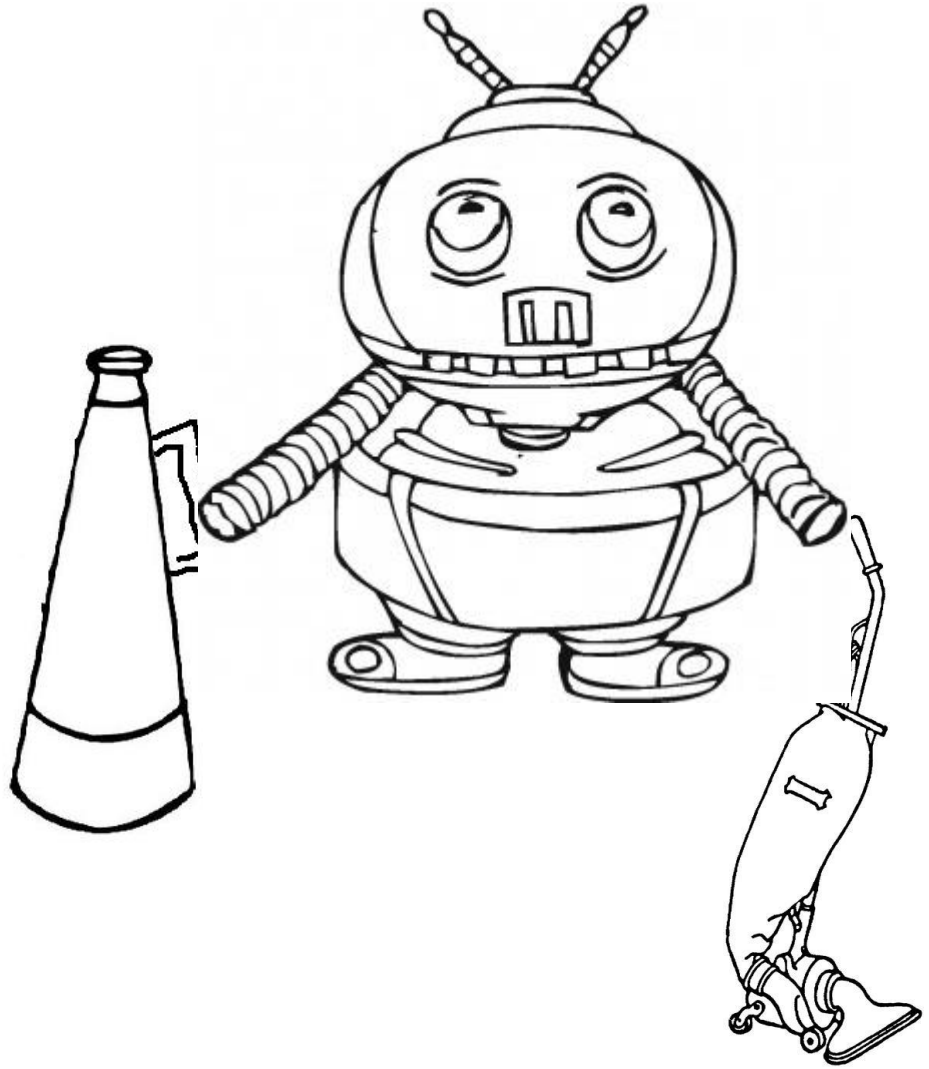
Agenda – you will choose

- Spring + JPA
- Spring + RMI
- Spring + Quartz
- Spring Data
- Spring MVC
- Spring Security
- Spring Validations
- Spring Jdbc
- Spring Boot
- Spring Data
- Spring Rest
- Spring Cloud
- …

How do you create an object?

1. Use new keyword
2. I’m a superman, I use only reflection
3. Use a service which creates object
4. I don’t create objects, I use only static methods.

What’s wrong with **new**?

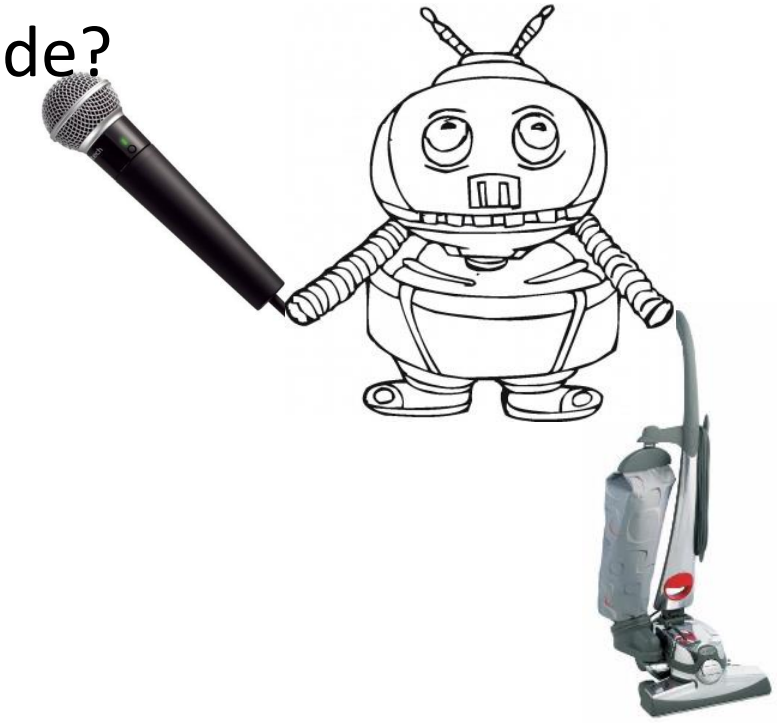


```
public class IRobot {
    private VacuumCleaner vacuumCleaner = new VacuumCleaner();
    private Speaker speaker = new Speaker();


    public void cleanDustInTheRoom(RoomFrame room) {
        speaker.sayJobStarted();
        vacuumCleaner.suck();
        room.makeRoomBrighter();
        speaker.sayJobFinished();
    }
}
```

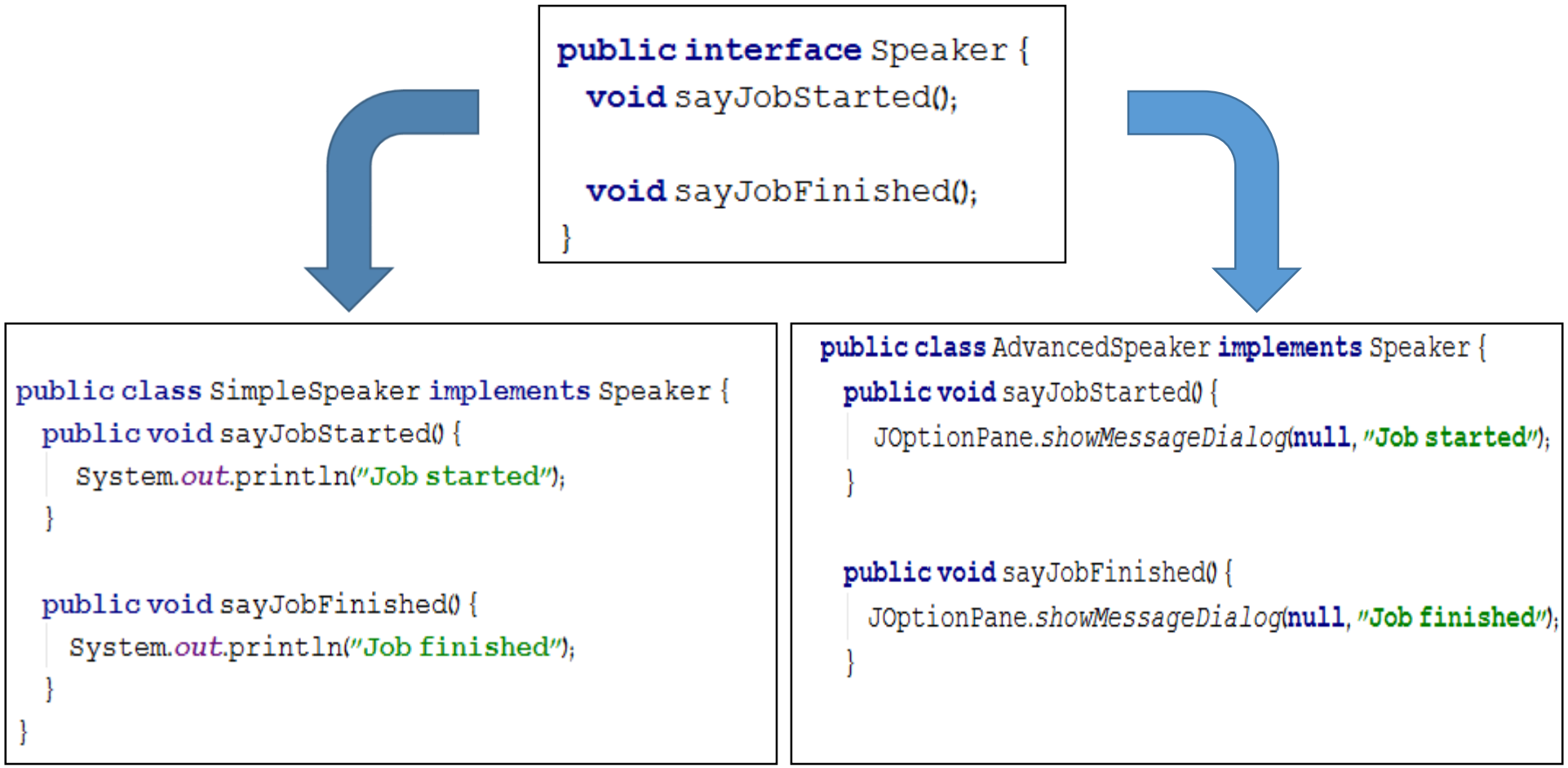
Any problems?

```
private VacuumCleaner vacuumCleaner = new VacuumCleaner();
private Speaker speaker = new Speaker();
```

- What if we need another Speaker implementation?
- We need change in the code?



Interface is better, isn't it?

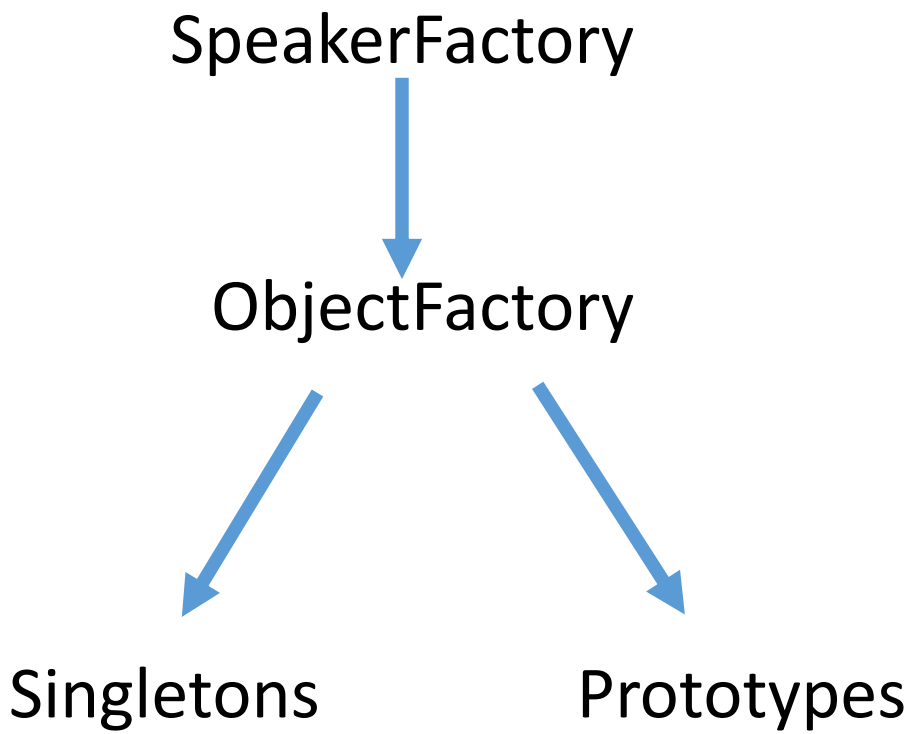


```
public interface Speaker {
  void sayJobStarted();

  void sayJobFinished();
}
```

```
public class SimpleSpeaker implements Speaker {
  public void sayJobStarted() {
     System.out.println("Job started");
  }

  public void sayJobFinished() {
    System.out.println("Job finished");
  }
}
```

```
public class AdvancedSpeaker implements Speaker {
  public void sayJobStarted() {
    JOptionPane.showMessageDialog(null, "Job started");
  }

  public void sayJobFinished() {
    JOptionPane.showMessageDialog(null, "Job finished");
  }
```

Who will decide which impl to choose?



Can you write a singleton?

6 phases of being familiar with singleton:

Phase 1:
«a Student»

Phase 2: «Juniour»

What about multithreading?

```
public class SpeakerFactory {

  public static SpeakerFactory speakerFactory;

  private SpeakerFactory() {
  }

  public static SpeakerFactory getInstance() {
    if (speakerFactory == null) {
      //create new factory and initialize all it's stuff
      speakerFactory = new SpeakerFactory();
    }
    return speakerFactory;
  }

  public Speaker createSpeaker(){
    //do all business logic
    return new SimpleSpeaker();
  }
}
```

Phase 3: «Middle Software Engineer»

What about performance?

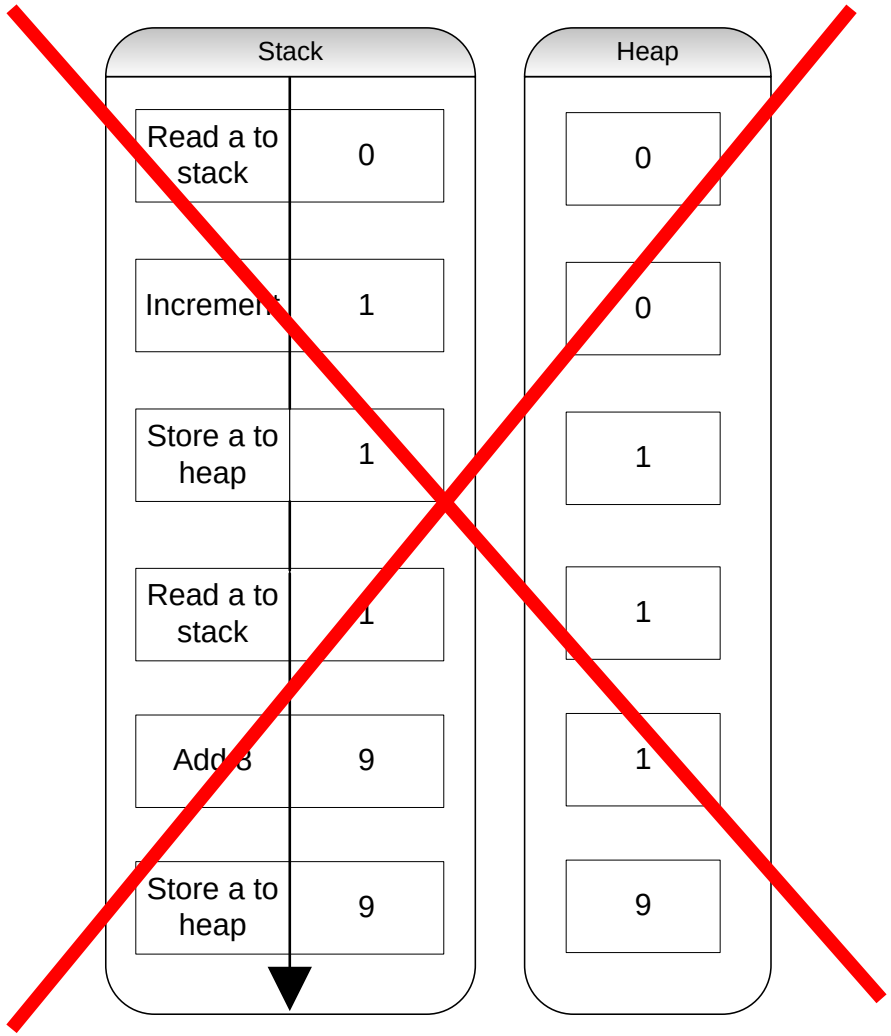
```
public class SpeakerFactory {

  public static SpeakerFactory speakerFactory;

  private SpeakerFactory() {
  }

  public synchronized static SpeakerFactory getInstance() {
    if (speakerFactory == null) {
      //create new factory and initialize all it's stuff
      speakerFactory = new SpeakerFactory();
    }
    return speakerFactory;
  }

  public Speaker createSpeaker(){
    //do all business logic
    return new SimpleSpeaker();
  }
}
```

Phase 4: «Senior Software Engineer»

What about java optimization?

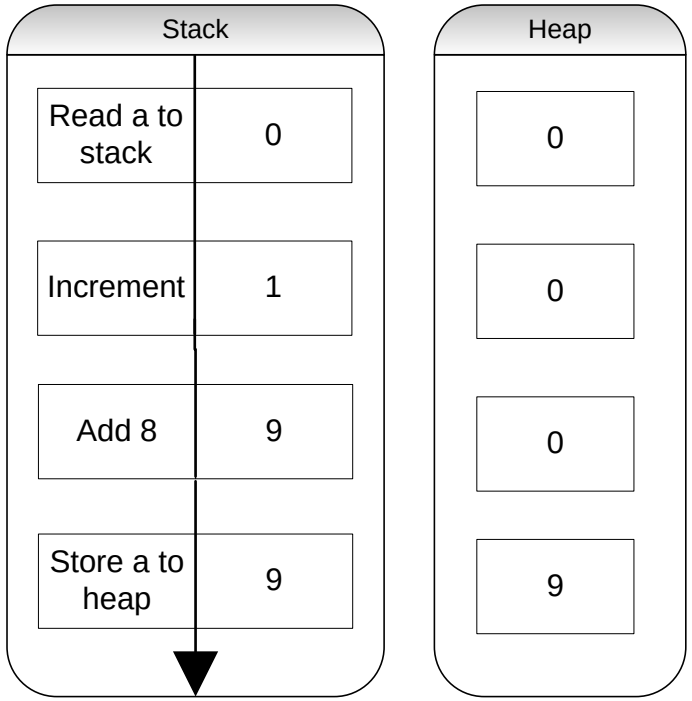
```
public class SpeakerFactory {

    public static SpeakerFactory speakerFactory;

    private SpeakerFactory() {
    }

    public static SpeakerFactory getInstance() {
        if (speakerFactory == null) {
            synchronized (SpeakerFactory.class) {
                if (speakerFactory == null) {
                    //all needed stuff for initialization
                    speakerFactory = new SpeakerFactory();
                }
            }
        }
```

Out-Of-Order Execution




```
 1 public class OutOfOrder {
 2
 3     private int a;
 4
 5     public void foo(){
 6         a++;
 7         a+=8;
 8     }
 9
10 }
```

Out-Of-Order Execution




```
 1 public class OutOfOrder {
 2
 3     private int a;
 4
 5     public void foo(){
 6         a++;
 7         a+=8;
 8     }
 9
10 }
```

Phase 4: «Senior Software Engineer»

What about java optimization?

```
public class SpeakerFactory {

    public static SpeakerFactory speakerFactory;

    private SpeakerFactory() {
    }

    public static SpeakerFactory getInstance() {
        if (speakerFactory == null) {
            synchronized (SpeakerFactory.class) {
                if (speakerFactory == null) {
                    //all needed stuff for initialization
                    speakerFactory = new SpeakerFactory();
                }
            }
        }
```

Phase 5: «Lead Software Engineer»

```
public class SpeakerFactory {

    public volatile static SpeakerFactory speakerFactory;

    private SpeakerFactory() {
    }

    public static SpeakerFactory getInstance() {
        if (speakerFactory == null) {
            synchronized (SpeakerFactory.class) {
                if (speakerFactory == null) {
                    //all needed stuff for initialization
                    speakerFactory = new SpeakerFactory();
                }
            }
        }
        return speakerFactory;
    }
```

Why you shouldn’t write a singleton

- You can not trust the future
- Concentrate on your business logic instead of reinventing the wheel
- How will you test?

Phase 6 «Architect»

- Don’t write Singletons.
  You have spring!

Why Spring?

- Many design patterns out of the box
  - Strategy, Factory, Singleton, Proxy
- Dependency injection
- You can test (or run) without application server
- Integration with all good frameworks
  - Hibernate, Quartz, JMS, RMI, WEB…
- AOP
- Spring mature

The history

- When the developers hated EJB 2 and J2EE 1.3
- In the time when in order to write EJB you was needed to create 1000 classes and xmls
- When Java was without annotations…
- In 2002 first Spring release

History

- First spring release in 2002
- Spring 1.0 in 2004
- In the end of 2005 spring become a leading J2EE platform
- 2009 VMware buy Spring framework
- 2013 Spring move to pivotal

What do we have today

- Spring 5.0+
- 4.3.+

Spring philosophy

- Inversion of Control (IoC)
- Dependency Injection
- J2EE must be simple
- Interfaces!!!
- Java conventions very strong
- OOP!
- No more checked exceptions!
- Tests are very important

IoC examples?

- Applet, Midlet
- Servlets instantiation & life-cycle.
- EJBs life-cycle.
- Event-driven applications.
- **Dependency Injection**.

Dependency Injection

- Solves the coupling problem
- Coupling – It means that if, for example, module A is coupled to module B, A **can't** be used without B.

```
public class IRobot {
  private VacuumCleaner vacuumCleaner = new VacuumCleaner();
  private Speaker speaker = new Speaker();


  public void cleanDustInTheRoom(RoomFrame room) {
    speaker.sayJobStarted();
    vacuumCleaner.suck();
    room.makeRoomBrighter();
    speaker.sayJobFinished();
  }
}
```

Configure Spring Context with XML

```xml
<?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">

</beans>
```

What is bean?

What is bean (convention)

- **SpringBeans** are reusable software components for Java.
- Spring bean is any java class
  - Can be a service
  - Can be POJO
  - Can be our class
  - Can be 3-d party library class

Defining beans in xml

```
<!--Devices-->
<bean id="speaker" class="devices.RussianSpeaker" scope="prototype"/>
<bean id="vacuumCleaner" class="devices.VacuumCleaner" scope="prototype"/>
```

```
<bean class="robots.IRobot" name="iRobot">
  <property name="speaker" ref="speaker"/>
  <property name="vacuumCleaner" ref="vacuumCleaner"/>
</bean>
```

How can you give a name to bean in xml

- Attribute ID
- Attribute name
- Use tag <alias> if you want to add additional name to already defined bean
- `<alias name="fromName"alias="toName"/>`

More basic bean definitions

- Attribute “class” – takes fully qualified class name
- Default scope = singleton
- By default all singletons will be created as part of a context
- In order to change: <beans…> default-lazy=true
- For specific bean: lazy=true

How does property tag work?

- Property – means, that you have a setter. (java conventions)
- So when you know the name of the property, setter name can easily be guesed: setPropertyName(…)
- Than setter is invoked and receives any parameter you transfer

Stop! How deep you familiar with Reflection?

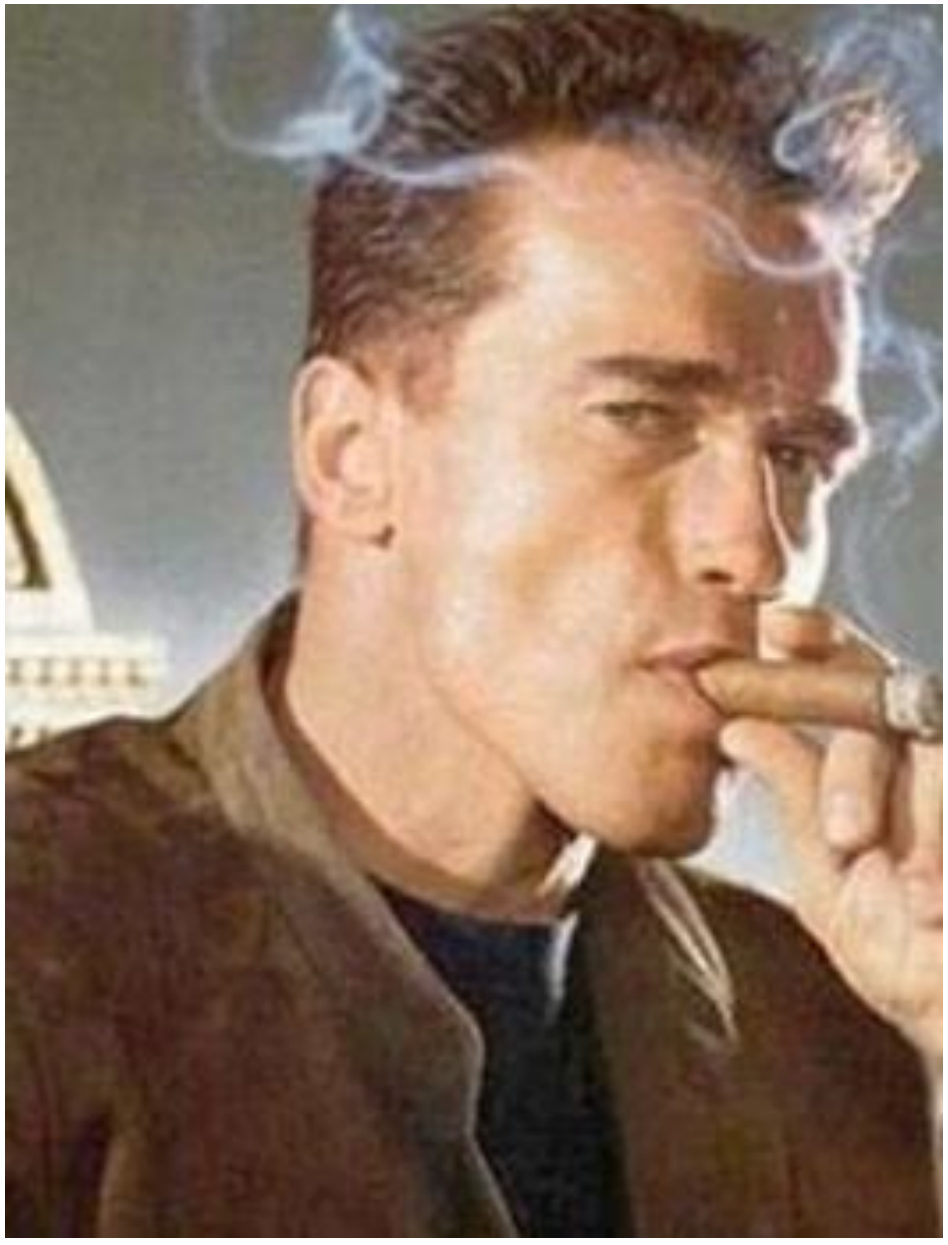
What is you level?

1. Reflection? Never heard of it
2. I know what is that
3. I’m using it
4. I wrote my own custom annotations and scanned them in runtime
5. I know the difference between RetentionPolicy.RUNTIME, RetentionPolicy.SOURCE и RetentionPolicy.CLASS
6. I know the what is the default retention policy

Task

- Write your own annotation, lets say: RunThisMethod
- It must have repeat int parameter
- Write ObjectFactory with method createObject
- Method receives class type, and returns new instance of this type, but if this class has @RunThisMethod annotation above some methods, this methods must be invoked several times, according to repeat parameter
- Test your framework

Now you do know Reflection



What can be injected to spring beans?

1. Other beans. Use attribute “ref”
2. primitives and Strings. Use attribute value
3. Collections.
    1. List
    2. Set
    3. Map

Inject value to bean

```
<bean class="quoters.Person">
    <property name="firstName" value="Jack"/>
    <property name="age" value="35"/>
</bean>
```

Inject one bean to another

```
<!--Devices-->
<bean id="speaker" class="devices.RussianSpeaker" scope="prototype"/>
<bean id="vacuumCleaner" class="devices.VacuumCleaner" scope="prototype"/>
```

```
<bean class="robots.IRobot" name="iRobot">
    <property name="speaker" ref="speaker"/>
    <property name="vacuumCleaner" ref="vacuumCleaner"/>
</bean>
```

List injection

```
<property name="messages">
    <list value-type="java.lang.String">
        <value>Shalom</value>
        <value>Privet</value>
        <value>Hi</value>
        <value>Guten Tak</value>
    </list>
</property>
```

Who to get the bean from the context

Context: creations and usage

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(xmlName);
IRobot iRobot = (IRobot) context.getBean("iRobot");
iRobot.cleanDustInTheRoom(room);
```

- It called Lookup
- It is possible to make lookup by name of the bean, name of the class or interface
- Lookup can be used for testing
- Or for the integration with legacy frameworks, like strats 1
- And what about production?

Which kind of application are exists?

- Event Driven

1. Define all your beans
2. Inject all the dependencies
3. Start the spring context

- Scheduled

1. Define all your beans
2. Inject all the dependencies
3. In case your application have main method -> configure init method of the main bean
4. If you need a scheduler –> than define quartz job as spring bean

Task

- Write an interface Quoter with method sayQuote and 2 implementations
  - ShakeSpearQuoter (with property message) and TerminatorQuoter (messages)
- Method sayQuote will print all the messages of specific quoter
- Declare them as beans in context and inject all messages via xml
- Test
  - Create context and make a lookup
  - Try lookup by name of the interface.
  - Explain the exception

How to define init method of bean?

- In xml context there is attribute init-method
- You can use this attribute inside tag bean
- You can use default-init-methods in tag beans if you have some common init methods to all methods

Task – Inject both Shakespeare and Terminator to additional bean

Task

- Write interface and impl of TalkingRobot which will have property List<Quoter> and override method talk, that he will iterate al over each Quoter and delegate to their sayQuote method)
- Define this bean in xml and inject both qouters
- Define method talk, as init-method
- In your test you just create the context
- Change scope to prototype
- Why nothing is printed now?

Constructor Injection

- Inside bean use tag: <constructor-arg>
- Has attribute: type, index (starts from 0), name, value and ref

```
<bean class="Wizard">
      <constructor-arg index="1"  value="Серый"/>
      <constructor-arg index="0"  value="Гендальф"/>
</bean>
```

```
<constructor-arg value="12" type="int"></constructor-arg>
```

- Now it’s time for real magic

Task

- Create bean of type String and inject it as additional terminator quote

How else can I run the init-method?

- Init Method can define somebody who configure the context
- And what if developer want to declare it?

1. You can implement interface InitializingBean, and override afterPropertiesSet method

- But it is legacy way, before the annotations

2. You can use @PostConstruct above needed method

I have a question

1. What if I have more than one method annotated with @PostConstruct
2. Why not to use the constructor?

I have a riddle, even 2

```
public class Parent {

    public Parent() {
        printPi();
    }

    public void printPi(){
        System.out.println("Pi");
    }
}
```

```
public class Son extends Parent {
    private double pi = Math.PI;

    public Son() {
        printPi();
    }
    @Override
    public void printPi() {
        System.out.println(pi);
    }
}
```

```
public static void main(String[] args) {
    Son son = new Son();
}
```

?

Answer:

```
0.0
3.141592653589793
```

Can you order this cycles

- @PostConstruct
- Spring setter (annotation) injection
- Son Initializer
- Parent Initializer
- Son inline
- Parent Inline
- Son Constructor
- Parent Constructor

The correct order

- Parent Inline
- Parent Initializer
- Parent constructor
- Son Inline
- Son Initializer
- Son constructor
- Spring setter (annotation) injection
- @PostConstruct / init=mathids

Why to know all this!!! To pass the interview?

Practice usage

```
public class BestService {
    public BestService(){
        chuckNorrisMethod();
    }

    public void chuckNorrisMethod() {
        out.println("Save the world");
    }
}
```

```
public class BetterThanBestService extends BestService {
    private List cache = new ArrayList();

    @Override
    public void chuckNorrisMethod() {
        cache.add(1);
    }
}
```

```
public static void main(String[] args) {
    new BetterThanBestService();
}
```

?

1. Will not compile.
2. Runtime exception.
3. You can't override Chuck Norris methods.
4. Everything is ok.

```
    public TerminatorQuoter() {
        sayQuote();
    }
    @Override
    public void sayQuote() {
        for (String message : messages) {
            System.out.println(message);
        }
    }
}
```

What will happen now?

```
public class TerminatorQuoter implements Quoter {
    private List<String> messages = Arrays.asList("полюбому");

    @Override
    @PostConstruct
    public void sayQuote() {
        for (String message : messages) {
            System.out.println(message);
        }
    }
}
```

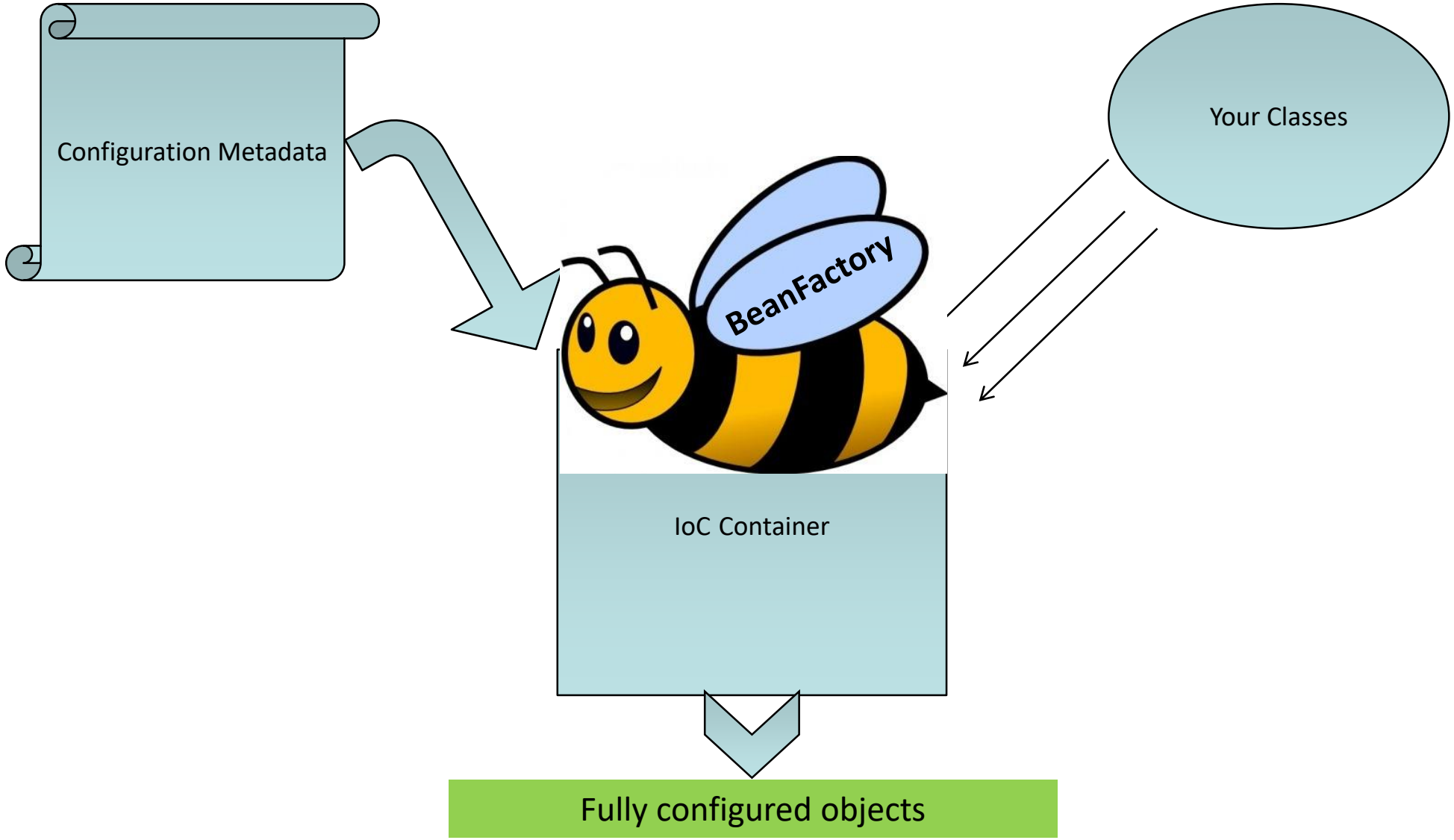
Let’s try @PostConstruct now

- Why the @PostConstruct is ignored?
- XML context not scan annotations by default
- We need to introduce additional spring internal components
- Add this bean to your context:CommonAnnotationBeanPostProcessor
- See what will happen

4 levels of understanding of BeanPostProcessor

- I can pronounce it
- I understand how do they work
- I can write simple BeanPostProcessor
- I can write BeanPostProcessor which creates Proxy

How does it work?



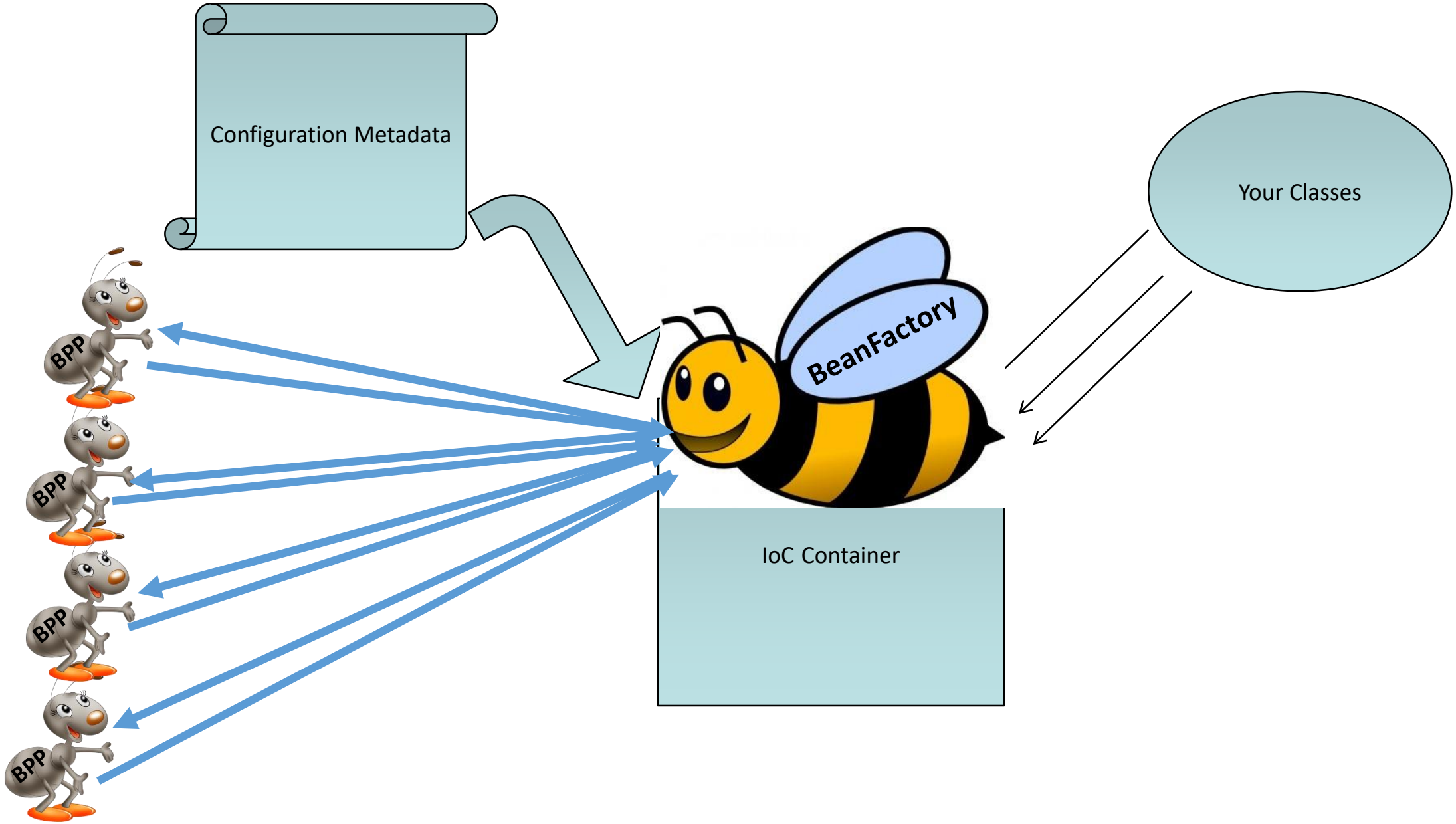
And what about BeanPostProcessors?

- BeanFactory creates and configures object according to metada which parsed from xml by BeanDefenitionDocumentReader
- All beans which implements interface BeanPostProcessors are not applicative beans, but part of spring framework
- They take a part of configuring applicative beans
- So BeanFactory will create them first

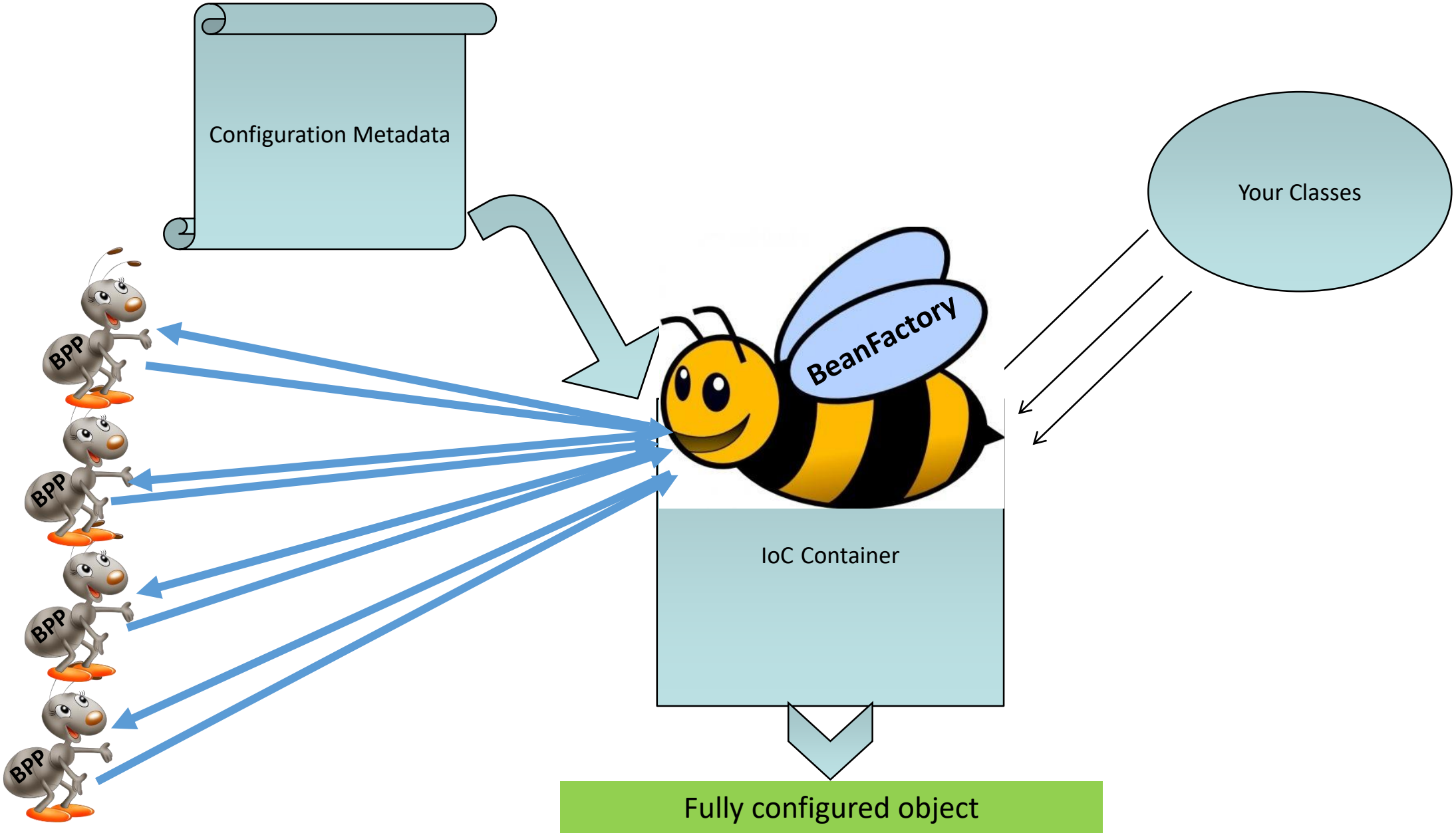
BeanPostProcessor interface

- Object postProcessBeforeInitialization(Object bean, String beanName)
- This method will be called by BeanFactory per each applicative bean. After it was created and all dependencies was inject, but before init method
  Object postProcessAfterInitialization(Object bean, String beanName)
- This method will be invoked by BeanFactory after init method

Before Init Method



After Init Method



Task

- Write your own BeanPostProcessor which will care about custom RunThisMethod annotation

Task

- Write BeanPostProcessor which will handle custom annotatiom @InjectRandomInt
- This annotation will have to parameters: min & max

Task

- Write BeanPostProcessor which will handle custom annotatiom @Benchmark
- Every methods of class annotated with @Benchmark must be wrapped with benchmark logic and the result must be printed to the log

- Improve this functionality. Now the annotation must be beyond the method and not the class

2 ways to create proxy

- java.lang.reflect.Proxy.newProxyInstance
- CGLIB jar

Very complicated task

- Write additional BeanPostProcessor which also will create a proxy.
- He will handle all beans annotated with @Transaction and will wrap all methods with transactions
- Test it…

THERE CAN BE ONLY

ONE BEANPOSTPROCESSOR

Destroy Method

- Finalize – not for use
- Where to put the destroy logic?
- You can declare destroy method in any bean
- Use destroy-method attribute in bean tag
- Or default-destroy-methods in tag beans
- Destroy methods work when context closing
- Destroy method never works for prototypes

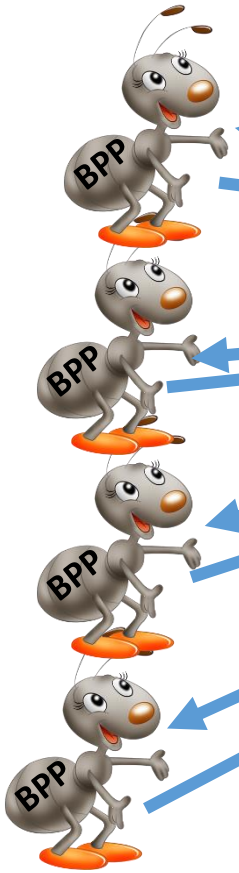
Task

- Declare additional spring component which will find all prototypes with destroy method and will print a warning about such cases

BeanFactoryPostProcessor

Configuration Metadata

Your Classes



IoC Container

Fully configured objects

BeanFactoryPostProcessor

- This is interface with only one method:
- postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
- This method will work before beans BeanFactory starts the bean creation, and there are only BeanDefenitions exists
- This is an entry point if you want to read or change the metadata of BeanDifinitions
- Now you can complete the task

Another task

- Write your own annotation which will extend the @Deprecated
- This annotation will receive parameter of newClass implementation
- Write BeanFPP which will change all implementations of class annotated with @MyDeprecated with class definition declared in this annotation parameter

Existing BeanFactoryPostProcessor

- If you want to inject values from property files to xml you need to use:

```
<context:property-placeholder location="aplication.properties,
```

- Location can be: classpath / file / http

```
<bean class="properties.example.Login">
    <constructor-arg name="name" value="${name}"/>
    <constructor-arg name="password" value="${password}"/>
</bean>
```

Factory Bean

- If you configure your context via XML, how can you force spring to create beans not with constructors, but with your custom logic?

What do you think about that?

```
<bean id="guestroom" class="ui.RoomFrame" scope="prototype">
 <property name="scuba" ref="scuba"></property>
 <property name="IRobot" ref="iRobot"></property>
 <property name="duster" ref="duster"></property>
 <constructor-arg value="Guest Room"></constructor-arg>
</bean>

<bean id="hall" class="ui.RoomFrame">
 <property name="scuba" ref="scuba"></property>
 <property name="IRobot" ref="iRobot"></property>
 <property name="duster" ref="duster"></property>
 <constructor-arg value="Hall"></constructor-arg>
</bean>
```

Beans inheritance

```
<bean id="room" class="ui.RoomFrame" abstract="true">
 <property name="scuba" ref="scuba"></property>
 <property name="IRobot" ref="iRobot"> </property>
  <property name="duster" ref="duster"> </property>
</bean>

<bean id="hall" parent="room">
 <constructor-arg value="Hall"></constructor-arg>
</bean>

<bean id="guestroom" parent="room‟scope="prototype">
 <constructor-arg value="guestroom"/>
</bean>
```

XML inheritance

Inside your xml context you can just…

```
<import resource="english-application-context.xml"/>
```

Annotations

- @PostConstruct / @Predestroy
- @Autowired / @Inject / @Resource
- @Qualifier
- @Component / @Service / @Repository
- @Controller
- @Scope
- @Lazy

Should we remember all BeanPostProcessors names, which responsible for this annotations???

2 ways to activate annotations

```
    <context:component-scan base-package="lesson4"></context:component-scan>
</beans>
```

Only one line

```
<context:component-scan base-package="com.inwhite.services">
        <context:include-filter type="regex"
                                expression="com\.inehite\.services\..*Course"/>
        <context:exclude-filter type="annotation"
                                expression="com.inwhite.annotations.Deprecated"/>
</context:component-scan>
```

Or:

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext("lesson4");
```

How to declare beans via annotations

- ```
  @Component
  public class ShakeSpearQuoter {
  ```
- ```
  @Component("tQ")
  public class TerminatorQuoter {
  ```
- ```
  @Service
  public class MyService {
  ```
- ```
  @Repository
  public class MyDao {
  ```

Dependency Injection annotations

- @Resource – javax
- @Inject - javax
- @Autowired - spring
- @Value – spring

- Injection to static fields is not supported

@Resource – Injection by the name of the bean

- ```
  @Resource
  private DbService dbService;
  ```
- Injects the bean with name: dbService
- And what the name of bean differs from property name?
- ```
  @Resource(name = "databaseService")
  private DbService dbService;
  ```
- @Resource can be used beyond fields or setters
- Handled by CommonAnnotationBeanPostProcessor
- @Resource better not to use

Difficult choice

@Autowired

@Inject

@Autowired

- Spring annotation
- Can be used beyond field, setter or constructor
- Injects by type and if more than one fails
- Default @Autowired(required=true), but you can change
- Use @Qualifier in order to be more specific

@Inject

- Java standard - advantage
- Can’t be configured with required attribute (required is always true)
- @Autowired(required=true) = @Autowired = @Inject
- @Autowired(required=false) = @Autowired(required=false)

Difficult choice

@Autowired

tasty

@Inject

standard

@Qualifier

- Meta annotation
- Can be combined with @Autowired

```
@Autowired
@Qualifier("terminatorQuote")
private String message;
```

- But best practice is:
-

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

Usage:

```
@Autowired
@TerminatorQuote
private String message;
```

Another way:

```
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Quote {
    QuoteType value();
}
```

```
@Autowired
@Quote(QuoteType.)
private String
```

SHAKE_SPEAR QuoteType

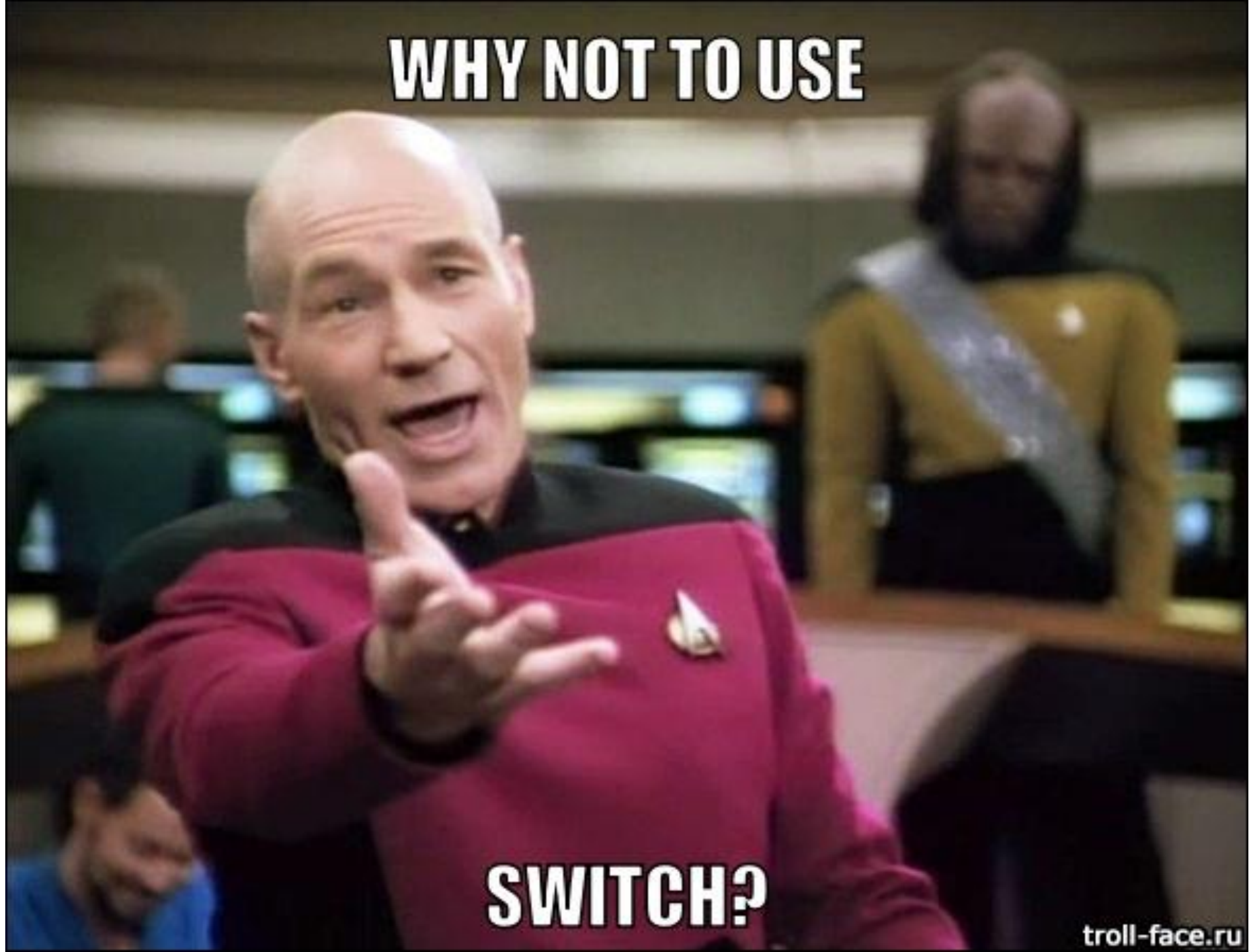
TERMINATOR QuoteType

Task

- Write to annotations: @Oracle и @Derby
- Write 2 implementation for interface Dao which overrides method crud()
- One class is oracle implementation and another is derby implementation
- Declare it with your own annotations
- Write Service, with property Dao and try to inject to it, first the oracle implementation and second the derby implementation

Write your own framework with spring

- Your application need to take some int parameter and run appropriative business logic.
- Lets start from 2 different options for this int.
- The code you will write now will become internal infrastructure.
- All your descendants will continue doing the same


WHY NOT TO USE
SWITCH?
troll-face.ru

We love you, Switch …

```
public DistribHandler resol
    switch (Integer.valueOf(ac
switch (documentObject.getDocumen
  case PDF_STORAGE:
    fileContainer = new PdfRecordFi
    getPdfFromStorage(fileContainer,
    break;
  case PDF_SRC:
    fileContainer = new PdfRecordFi
    fillObjectsForPdf(fileContainer, d
    break;
  case PDF_WS:
    fileContainer = new WsPdfRecor
    getPdfFromPdfWs(fileContainer, (
    break;
  case LIS:
    fileContainer = new PdfRecordFi
    getPdfFromLisDocument(j, fileCo
    break;
  case IMAGE:
    fileContainer = new PdfRecordFi
    getPdfFromImageDocument(j, file
    break;
  case FORM:
    fileContainer = new PdfRecordFi
    getPdfFromFormDocument(fileCo
    break;
}
```

```
switch (value.getNumericValue()) {
  case 1:
    text = MessageFormat.format("{0} הצעה לפוליסת", emailRequest.getSubject());
    break;
  case 2:
    text = MessageFormat.format(" {0} רבישת פוליסת", emailRequest.getSubject());
    break;
  case 3:
    text = MessageFormat.format(" {0} חידוש פוליסת", emailRequest.getSubject());
    break;
  case 4:
    text = MessageFormat.format("{0} שינויים בפוליסת", emailRequest.getSubject());
    break;
  case 5:
  case 8:
    text = MessageFormat.format("{0}מבתב מ", brandHebName);
    break;
  case 6:
  case 7:
    text = MessageFormat.format("{0}מבתב חשוב עבורך מ", brandHebName);
    break;
  case 9:
  case 17:
    text = MessageFormat.format("{0} - רבישת ביטוח נסיעות", brandHebName);
    break;
  case 10:
  case 13:
    text = MessageFormat.format("{0} - {1}", emailRequest.getSubject(), brandHebName);
    break;
  case 11:
    text = emailRequest.getSubject();
    break;
  case 12:
    if (!resources.getBrandKey().isBituhYashir()) {
      text = format("s%-תודה מ", brandHebName);
    } else {
      text = format("s%תודה מ", brandHebName);
    }
    break;
  case 14:
    text = MessageFormat.format("{0} - ביטוח תאונות אישיות - השקט הנפשי שלך", brandHebName);
    break;
  case 15:
    text = MessageFormat.format("{0} בקשה לצירת קשר לחידוש ביטוח", emailRequest.getSubject());
    break;
  case 16:
```

```
icyDatFileConsumer.class);
ll && item.getAddresseeCode().length() >= 1
de())) ? item.getAddresseeCode().trim() : "0";
dressee(addressee);
urrentPath, basket, dataConsumer);

icyDetailsConfConsumer.class);
urrentPath, basket, dataConsumer);

icyLetterConsumer.class);
lr(basket.getMetaDBasket().getPrtNr());
urrentPath, basket, dataConsumer);

icyCompulsoryConsumer.class);
etSeqNr(item.getCompSeqNr());
urrentPath, basket, dataConsumer);
```

Switch replacement example

```
switch (personalMailBusinessModel.getFtlType()) {
  case WELCOME:
    ftlService.fillWelcomeFtl(data, personalMailBusinessModel);
    break;
  case EMAIL_CALLBACK:
    ftlService.fillEmailCallbackFtl(data, personalMailBusinessModel);
    break;
  case PROTECTION_APPROVAL_REMINDER:
    ftlService.fillProtectApprovalReminderFtl(data, personalMailBusinessModel);
    break;
  case CALL_ROUTER:
    ftlService.fillCallRouterFtl(data, personalMailBusinessModel);
    break;
  case PREV_YEAR_INTERNET:
    ftlService.fillPrevYearInternet(data, personalMailBusinessModel);
    break;
  case PREV_YEAR_SALES:
    ftlService.fillPrevYearSales(data, personalMailBusinessModel);
    break;
  case PROPOSAL_PREMIUM_REMINDER:
    ftlService.fillProposalPremiumReminder(data, personalMailBusinessModel);
    break;
  case MORTGAGE:
    ftlService.fillMortgageFtl(data, personalMailBusinessModel);
    break;
  case MORTGAGE_ENSLAVED_STRUCTUR:
    ftlService.fillMortgageEnslavedStructurFtl(data, personalMailBusinessModel);
    break;
  case RENEWAL_WANT_TO_THINK:
    ftlService.fillRenewalWantToThinkFtl(data, personalMailBusinessModel);
    break;
  case PREV_YEAR_OUT_SALES_APARTMENTS:
    ftlService.fillPrevYearOutSalesApartmentsFtl(data, personalMailBusinessMode
    break;
  case LIFE_OUT_SALES_FOLLOW_UP:
    ftlService.fillLifeOutSalesFollowUpFtl(data, personalMailBusinessModel);
    break;
  case APPS_PROPOSAL_FOLLOW_UP:
```

```
ftlMainService.generateHtml(data, personalMailBusinessModel);
```

What can not be done with annotations

- Declare beans from 3rd party libs
- Declare more than one bean from the same class
- Work with property files

What can not be done with XML

- Write a code inside it. If your beans needs some business logic as part of configuration it is super inconvenient to work with xml

Task

- Declare 2 beans: JFrame (singleton) and Color (prototype)
- Color will be injected to the frame and become his background
- Color must be random

Java Config

- Annotate your configuration class with @Configuration
- In order to declare beans, use @Bean
- ```
  @Bean
  @Scope(BeanDefinition.SCOPE_PROTOTYPE)
  public Color randomColor(){
      Random random = new Random();
      return new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));
  }
  ```
- You can create a context from java config files
- ```
  @ComponentScan(basePackages = "com.servies")
  ```
- ```
  @Configuration
  public class AppConfig {
  ```
  ```

How to inject from property files without XML

- ```
  @PropertySource("classpath:mails.properties")
  @ComponentScan(basePackages = "com.services")
  @Configuration
  public class AppConfig {…
  ```
- ```
  @Bean
  public static PropertySourcesPlaceholderConfigurer
  configurer() {
      return new PropertySourcesPlaceholderConfigurer();
  }
  ```
- ```
  public class ShakespeareQuoter implements Quoter {
      @Value("${shakeSpearQuote}")
      private String message;
  ```
  ```

Beans non direct dependencies

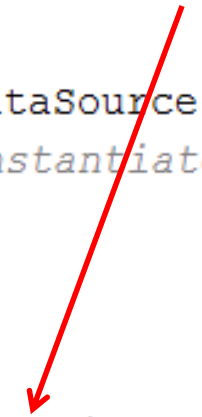
- Write 2 beans.
- First will create new file in his postconstruct
- Second need to print the length of the file from his postconstruct

@DependsOn

- Can be used in case there is no direct dependencies between beans

Combining several contexts

```java
@Configuration
public class DatabaseConfig {
    @Bean
    public DataSource dataSource() {
        // instantiate, configure and return DataSource
    }
}

@Configuration
@Import(DatabaseConfig.class)
public class AppConfig {
    @Autowired DatabaseConfig dataConfig;

    @Bean
    public DBService myBean() {
        // reference the dataSource() bean method
        return new DBService(dataConfig.dataSource());
    }
}
```



Xml + JavaConfig

```java
@Configuration
@ImportResource("classpath:/com/acme/database-config.xml")
public class AppConfig {
    @Autowired
    DataSource dataSource; // from XML

    @Bean
    public DBService myBean() {
        // inject the XML-defined dataSource bean
        return new DBService(this.dataSource);
    }
}
```

Lets go back to frame and color

- Make 50 lookups in your test and run frame method: showOnRandomPlace
- First scenario: frame and color are singletons
- Second scenario: both of them prototypes
- Third scenario: Frame is prototype, Color - singleton
- Fourth scenario: Frame is singleton, Color – prototype
- And now make color flick inside frame

Context types

Source/ Type	XML	Annotations
Classpath	ClassPathXmlApplicationContext	AnnotationConfigApplicationContext
File system	FileSystemXmlApplicationContext	
Web container	XMLConfigWebApplicationContext	AnnotationConfigWebApplicationContext

Comparing strategies

Bean Definition	XML	Annotations	@Configuration
How to declare?	<bean>	@Component class	@Bean method
What?	“class” attribute	Annotated class	You can’t know
Name	“id” or “name” attr	Decap. class name	Method name
Instantiation	Reflection	Reflection	Java code
Dependencies	<costructor-arg>, <property>	@Autowired	Constructor params, setter methods
Scope	“scope” attr	@Scope	@Scope
Init & destroy methods	<init-method>, <destroy-method>	@PostConstruct, @PreDestroy	Method invocation, “initMethod”, “destroyMethod” attributes

When does what suits

What is important to your project	XML	Annotations	@Config
Central configuration	✓	x	✓
I don’t want spring in my sources	✓	x	✓
No need to recompile when changing configuration	✓	x	x
IDE good support + refactoring	x	✓	✓
Minimize the configuration	x	✓	x

Comparing Config Strategies

Sample Usage	XML	Annotations	@Config
My source code		✓	
3rd-party utils and frameworks*			✓
Configuration by non-devs	✓		
Modular configuration requirements	✓		

*** Assuming that java is better programing language than XML**

Aspect Oriented Programming

The new paradigm

Aspect Oriented Programming

- What and Why
- Concepts and Terminology.

AOP Languages

- Just like we have OOP languages (Java, Groovy…) we also have AOP languages (AspectJ, Spring AOP, JBoss AOP, etc…).
- Before we dive into Spring AOP, we need to get familiar with the generic AOP concepts…

AOP instead of OOP?

No, it helps OOP

AOP

- As we are going to see soon, OOP doesn’t provide tools for ***cross-cutting-concerns***.
- In these areas, *Aspect Oriented Programming* can help.
- Examples: tracing, benchmarking, transactions, security, validation, logging, exception handling etc…

Tracing Feature Example

- For our feature we want to print a message (e.g. to *System.out*) at the beginning and end of every method.
- We want to print the method’s name, its argument values and return value (or exception).
- Now, lets say we have 50000 classes with 5 methods for each (on average).
- This means we need ~(50000*5*2) = 500000 tracing places.
- Nothing too hard. Right?

Main Concepts of AOP

- The main concepts of AOP are:
  - Aspect like class
  - Advice like method
  - Joinpoint.
  - Pointcut.

When advice should work

Types of Joinpoints

- With Spring, deals only with methods **method execution.**

Weaving

- Instrument bytecode in compile time
- Instrument bytecode in in bootstrap time
- Interceptors
- Proxy

Spring use AspectJ syntax

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
</dependency>
```

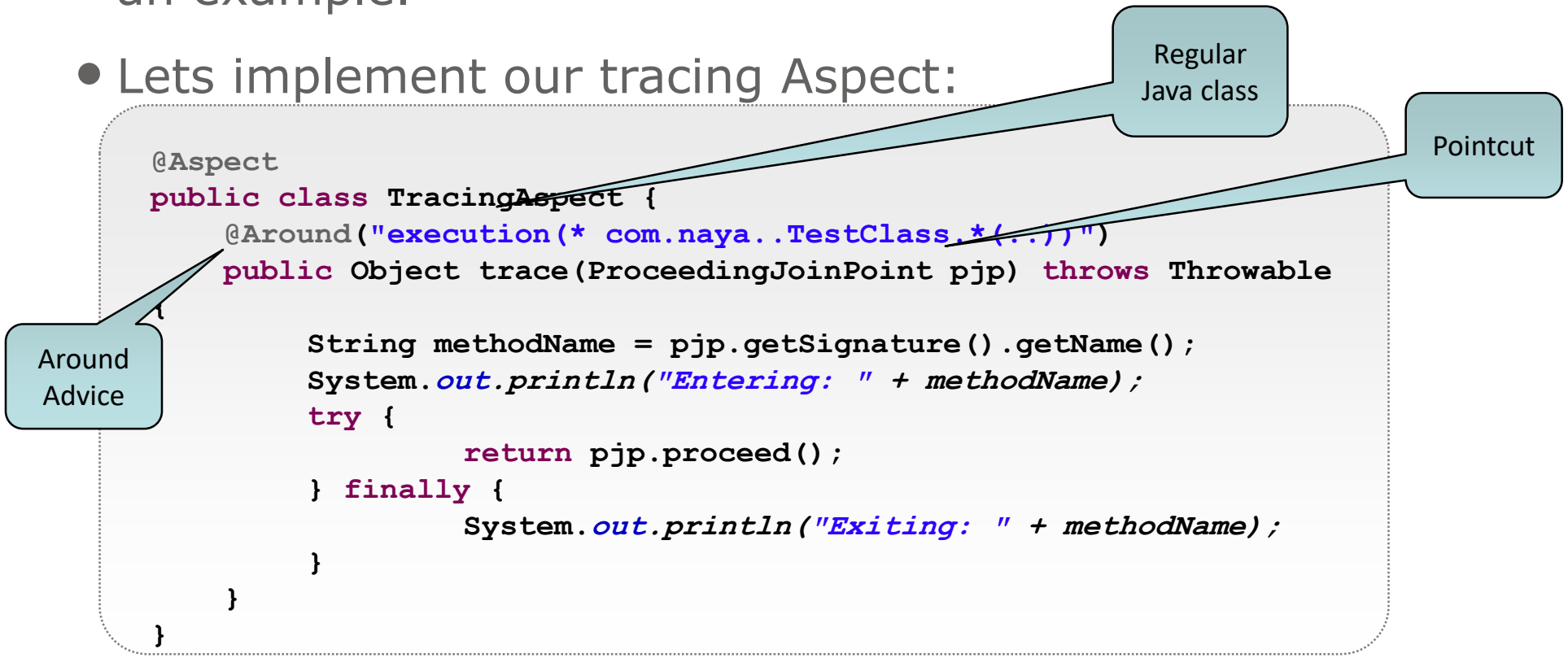
To make aspectJ annotation work

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

```
@EnableAspectJAutoProxy
public class AppConfig {
```

Spring AOP Example

- Lets see how to declare Aspect, Advice and Pointcut via an example.
- Lets implement our tracing Aspect:




```
@Aspect
public class TracingAspect {
    @Around("execution(* com.naya..TestClass.*(..))")
    public Object trace(ProceedingJoinPoint pjp) throws Throwable
    {
        String methodName = pjp.getSignature().getName();
        System.out.println("Entering: " + methodName);
        try {
            return pjp.proceed();
        } finally {
            System.out.println("Exiting: " + methodName);
        }
    }
}
```

Before Advice

- The ‘before’ Advice operates before the target method.
- The original method will be executed after the Advice is finished.
- Example:

```
@Before("execution(* com.naya..*.foo())")
public void beforeFoo(JoinPoint jp) {
            System.out.println("BEFORE foo()");
}
```

After Returning Advice

- Will operate after a normal execution of the method.
- Example:

```
        @AfterReturning(pointcut = "execution(*
com.naya..*.boo())",
                        returning = "retVal")
    public void afterBoo(Double retVal) {
        System.out.println("AFTER boo()" + retVal);
    }
```

- As you can see, we can access the return value as a parameter.

After Throwing Advice

- Will operate after an abnormal execution that threw an exception:

```
@AfterThrowing(pointcut = "execution(* com.naya..*.ex())",
                        throwing = "ex")
public void afterEx(IllegalArgumentException ex) {
         System.out.println(ex.getMessage());
}
```

- As you can see, we can access the exception as a parameter. (It also restricts the exception types).
- **If exception is caught – aspect won't work**

After (finally) Advice

- Will operate after any execution of the method (normal or exception):

```
@After ("execution(* com.naya..*.foo())”)
public void afterFoo() {
        System.out.println("After FOOOOOOO”);
}
```

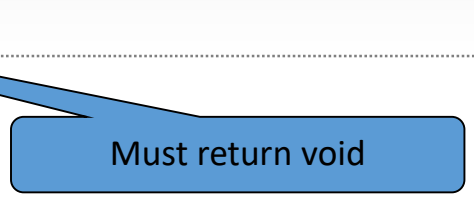
Around Advice

- This is the most generic type of advice.
- In this case, the Advice will be run **instead** of the original method.
- The signature of the Advice method must accept as a first parameter the type *ProceedingJoinPoint*.
- In order to invoke the original method (you don’t have to) just call the *proceed()* method.
- This advice hit the performance
- We have seen an example with the Tracing aspect.

Declaring Pointcuts

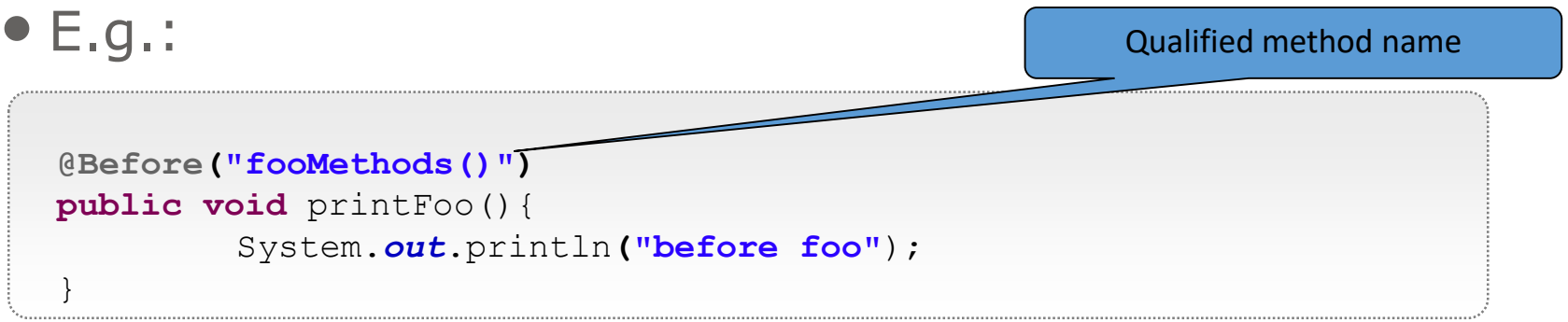
- We have seen examples of declaring pointcuts inside the advices, but for reusability, they can be declared separately.
- A pointcut is declared with the *@Pointcut* annotation and is placed on a method declaration.
- E.g.:

```
@Pointcut("execution(* com.naya..*.foo(..))")
private void fooMethods(){};
```



Using Pointcuts

- In order to use declared pointcuts in your advices, just use the method names on which they were declared.
- E.g.:



Pointcuts Syntax

- The syntax of the pointcuts in Spring is a **subset** of AspectJ syntax.
- Note, with Spring AOP, pointcuts will match **only public methods**.
- The general syntax of Spring AOP pointcut is:

  @Pointcut("**[pointcut designator](expr)**")

Pointcut Expressions

- The Pointcut expression (after the designator) resembles a regular expression.
- You can use the following wildcards:
  - ‘*’ – matches any character sequence without ‘.’.
  - ‘+’ – at the end of a **type** name includes all of its subclasses/implementations.
  - ‘..’ – matches a character sequence between ‘.’ and ‘.’ (recursive package names).

Examples

- To match all the methods inside all the classes that begin with package: *com.naya*:

```
@Pointcut("within(com.naya..*)")
private void allMethods(){};
```

- To match all the methods inside all the classes that begin with package: *com.naya*:

```
@Pointcut("allMethods() && @annotation(Deprecated)")
private void allDeprecated(){}
```

Combining another pointcut

Examples

- To match all the methods that accept a first parameter *String* and a second parameter *int*:

```
@Pointcut("allMethods() && args(a,b,..)")
private void useArgs(String a, int b){}

@Before("useArgs(a,b)")
public void printArgs(String a, int b){
        System.out.println(a);
        System.out.println(b);
}
```

Troubleshooting

Not working...

- Don't create Aspect in intelij, create regular class.
- Don't forget to annotate with @EnableAsptectJAutoProxy.
- @Aspect is above your aspect?
- Your aspect is introduced as spring bean? (@Component)
- Do you scan the package where is your aspect declared? Or do you use another way to declare your aspect?
- The methods you want to be intercepted by aspect, exists in spring beans, right?
- Don't you hope that aspect logic will run methods in init phase (like @PostConstruct)?

More Examples

execution(void send*(String))

Any method starting with send that takes a single String parameter and has a void return type

execution(* send(int, ..))

any method named send whose first parameter is an int (the “..” signifies 0 or more parameters may follow)

execution(void example.MessageService+.send(*))

any MessageService method named send that takes a single parameter and returns void

More Examples

execution(@Transactional void *(..))

Any method marked with the @Transactional annotation

execution((@privacy.Sensitive *) *(..))

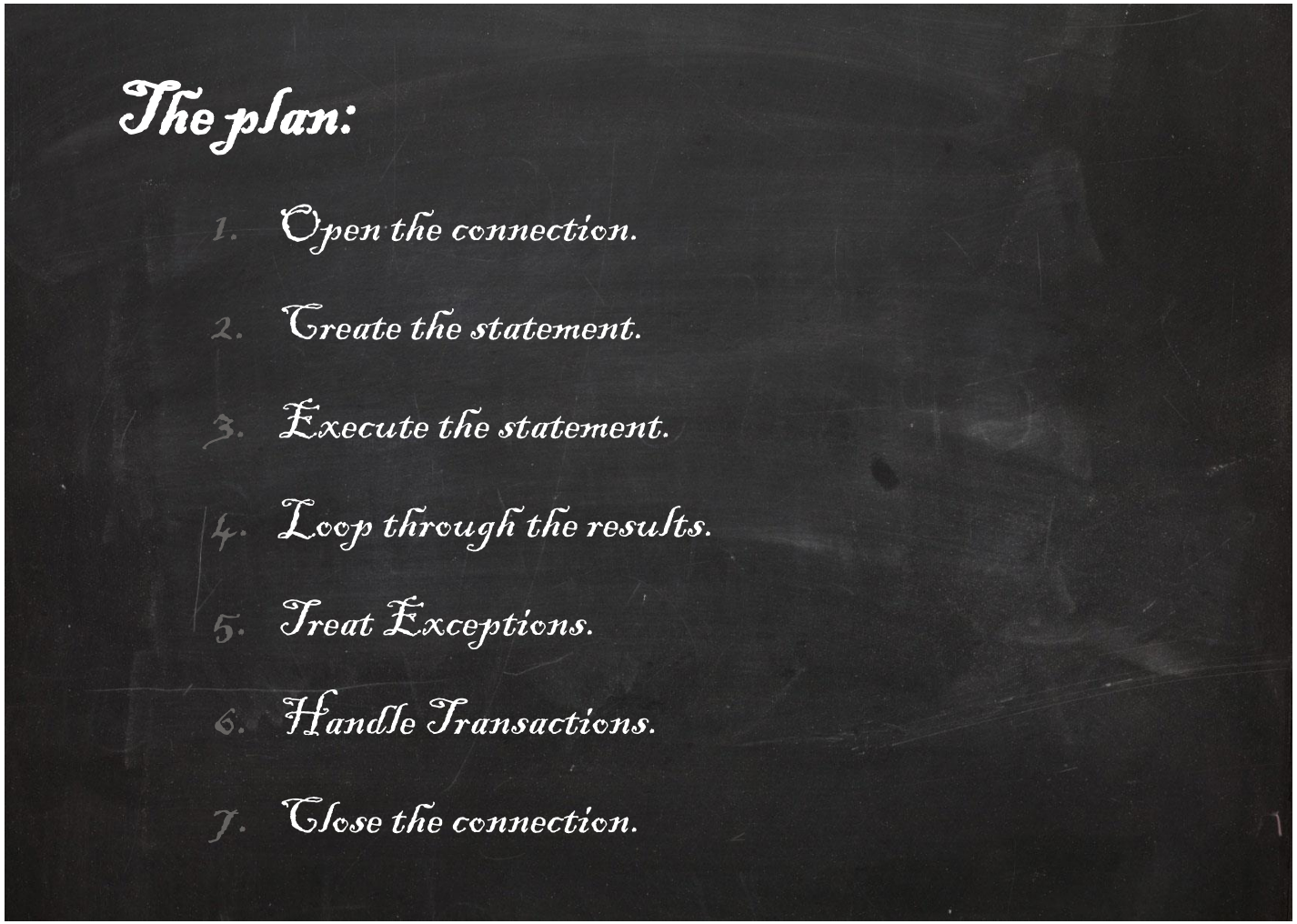
Any method that returns a value marked with the @Sensitive annotation

Data Access Using ORM

Data Access Using ORM

- ORM story
- Hibernate
- JPA story
- Spring ORM story
- Transactions

When we were young and worked with JDBC



The pain…

```
try {
    connection = DriverManager.getConnection("jdbc:…");
    PreparedStatement pStm = connection.prepareStatement(
    "Select TITLE from BOOKS where price < ?");
    pStm.setInt(1, 100);
    ResultSet rs = pStm.executeQuery();
    while (rs.next()) {
        System.out.println(rs.getString("TITLE"));
    }
} catch (SQLException e) {
        // What to do here? Throw again…
} finally {
    if (connection != null) {
        try {
                connection.close();
        } catch (SQLException e) {
                // to do…
        }
    }
}
```

The ORM Solution

- Mapping the object domain to a relational domain is a complicated task.
- Many projects have failed because they tried to develop an in-house persistence framework.
- The common solution today is to use a library, an **ORM** library.
- ORM = Object/Relational Mapping.

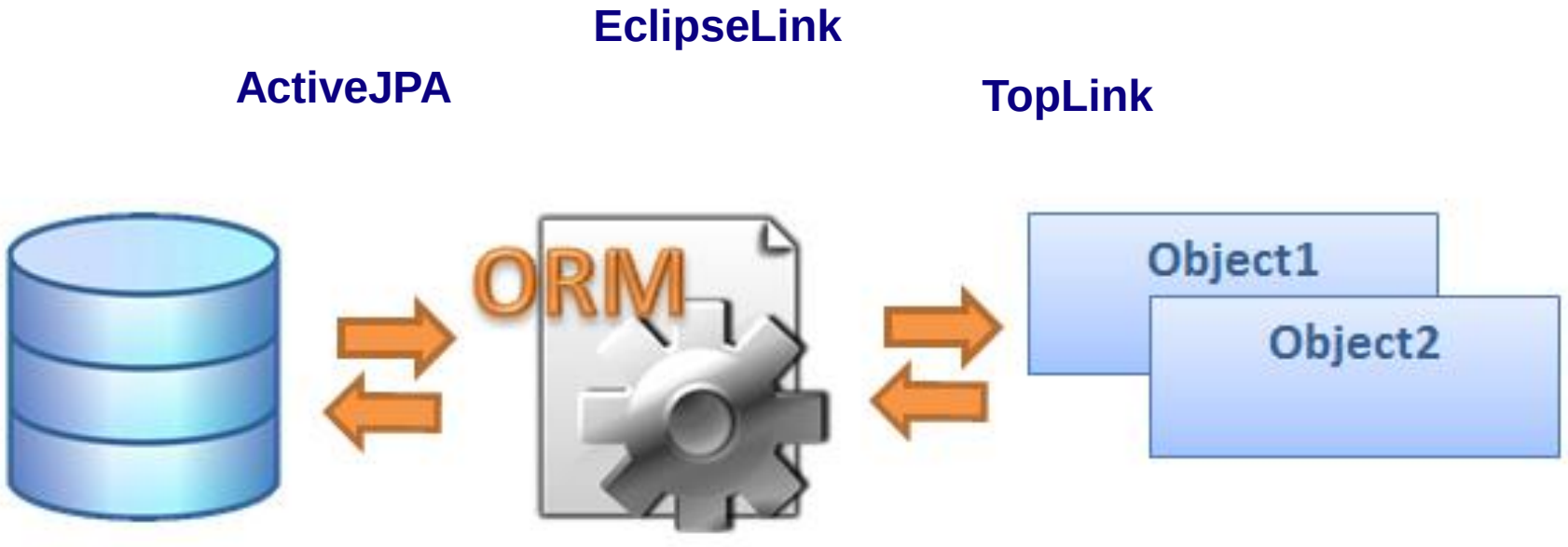
The Difficulties of ORM

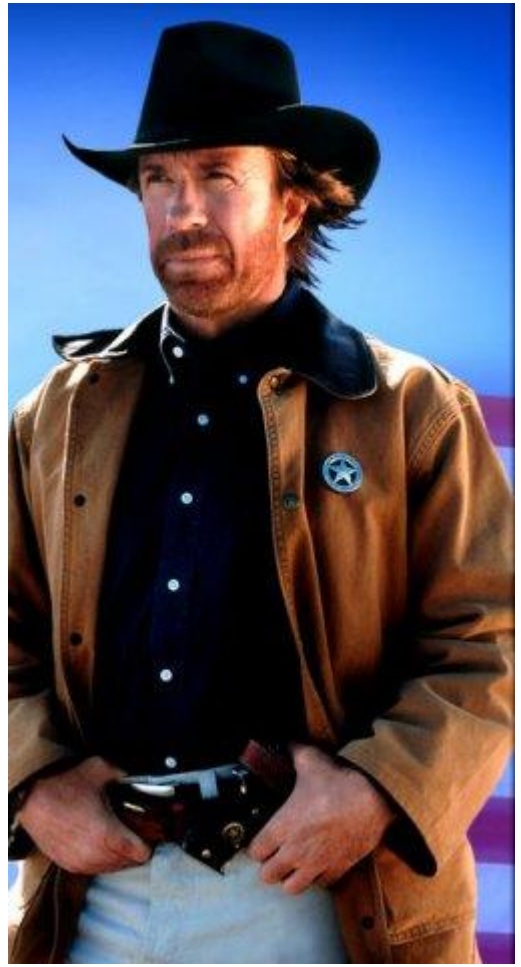
- ORM infrastructure is extremely difficult to develop.
- Challenges include:
  - Granularity – many classes are mapped to their own table, but some need to be embedded in other classes' table.
  - Lazy loading – when retrieving a collection from the DB, should we also retrieve nested collections?
  - Inheritance – how should it be handled?

ORM Software

- Thus, it is not recommended to develop your own ORM solution!
- Currently, there are many ORM libraries in the market, offering many features and great performance, e.g., OpenJPA, Hibernate, iBatis, EclipseLink, etc…

We should have some standard Lets call him JPA…





JPA History

- In the past (J2EE 1.4/EJB 2.1), the persistence solution was: Entity Beans.
- Entity Beans provided a poor ORM solution.
- Some major problems were:
  - No support for inheritance/ polymorphism.
  - Weak query language (e.g., no support for “group by”).
- Most projects didn’t use it!

Hibernate

- Many projects opted to using some non-standard ORM framework.
- The most popular ORM framework was Hibernate.
- The persistence solution of JEE 5 (JPA) is largely based on Hibernate.

Java Persistence API

- The JPA specification is part of the EJB3 specification (which is part of JavaEE 5).
- Starting from JavaEE 6 it is a standalone standard.
- Note that JPA can be used outside of an application server.
- JPA provides persistence framework for **POJO**s.
- JPA does not impose restrictions on your object model.

Basic Mapping Guidelines

- The general mapping idea is:
  - Map a class to a table.
  - Map a member/property to a column.
- Of course, there are many variations and exceptions to these guidelines.
- We will not cover all of the JPA standard.
- We'll just provide enough overview in order to understand JPA and then show what Spring provides on top.

Mapped Classes

- The mapped classes are POJOs.
- However there are still some requirements:
  - The class must have a no-arguments constructor that needs to be either public or protected.
  - The class must not be final and must not have final methods.

Persistent Fields

- Mapping can be done either by fields or by properties.
- Mapping by properties means that the JPA framework will use the getters/setters convention for the mapping.
- By default all the fields/properties are mapped.

- We will see how to control this in the next slides…

Mapping

- Mapping can done by annotations, XML files or both.
- Note, if both methods are used, XML configuration overrides the annotations.
- All configuration/mapping options can be done in both ways.

The Entity Annotation

- To state that a class is mapped in JPA, it must be annotated with **@Entity**.
- By default, the class will be mapped to a table that has the same name as the class.
- Also, the properties will be mapped to columns that have the same names.

Primary Key

- Each entity must have a primary key field/property.
- The annotation used to map it is: **@Id**.
- The primary key will be mapped to the primary key column in the DB.
- By default, the column name is the same as the mapped field.

Example

```
@Entity          // marking the class as being MAPPED
public class Company {
        private String name;
        private String symbol;
        private int id;

        public Company() {
                // required constructor for JPA
        }
        public Company(String name, String symbol) {
                this.name = name;
                this.symbol = symbol;
        }

        @Id       // marking the primary key
        public int getId() {return id;}
        public void setId(int id) {this.id = id;    }
        public String getName() {return name;}
        public void setName(String name) {this.name = name;}
        public String getSymbol() {return symbol;}
        public void setSymbol(String symbol) {this.symbol =
                                              symbol;}
}
```

Controlling The Table Name

- By default, the table name will be the same as the class name.
- In order to change this, use the **@Table** annotation.
- Important members of @Table:
  - name – the name of the table.
  - catalog.
  - schema.

Controlling The Column Name

- By default, the column name will be the same as the field/property name.
- In order to change this, use the **@Column** annotation.
- Important members of @Column:
  - name – the name of the column.
  - insertable – controls whether this column is included in the generated INSERT statement.
  - updatable – controls whether this column is included in the generated UPDATE statement.

Persistent Fields

- By default, all the fields/properties are mapped.
- In order to specify an **unmapped** field/property, use the **@Transient** annotation.
- If you are using field-access, the field will not be persisted if it is declared as *transient*.

Field Types

- According to the field type, some annotations may be present:
  - All persistent fields which are not a part of a relationship can be annotated with the **@Basic** annotation.
  - java.util.Date and java.util.Calendar types **must** be annotated with the **@Temporal** annotation.
  - Enums can be annotated with the **@Enumerated** annotation.

@Basic Annotation

- The @Basic annotation is optional.
- With the @Basic annotation you can provide the following members:
  - fetch – fetching mode, defaults to: Eager.
  - optional – controls whether the field can be null. Defaults to *true* (used for schema generation).

@Temporal Annotation

- The @Temporal annotation must be used for persistent fields of the types: Date & Calendar.
- Its value is either DATE, TIME or TIMESTAMP.
- It is used to map between a Java object and an appropriate column representation.

Surrogate or natural?

Primary Keys

- A primary key can be either simple (a single column) or composite (several columns).
- By default, the value for the primary key is assigned by the program (like any other field).
- However, the primary key can also be **generated**.
- This is done by using the **@GeneratedValue** annotation.

@GeneratedValue

- The @GeneratedValue annotation specifies that the primary key is generated.
- It has the following members:
  - startegy: can be one of *TABLE*, *SEQUENCE*, *IDENTITY* or *AUTO* (default).
  - generator: the name of the generator (used for SEQUENCE or TABLE).

Generation Type

- TABLE: the framework will use a special table for generating unique keys (used with @TableGenerator).
- SEQUENCE: the framework will use a sequence for generating unique keys (used with @SequenceGenerator).
- IDENTITY: the framework will use an auto-increment column.
- AUTO: the framework will choose the appropriate mechanism, depending on the underlying database.

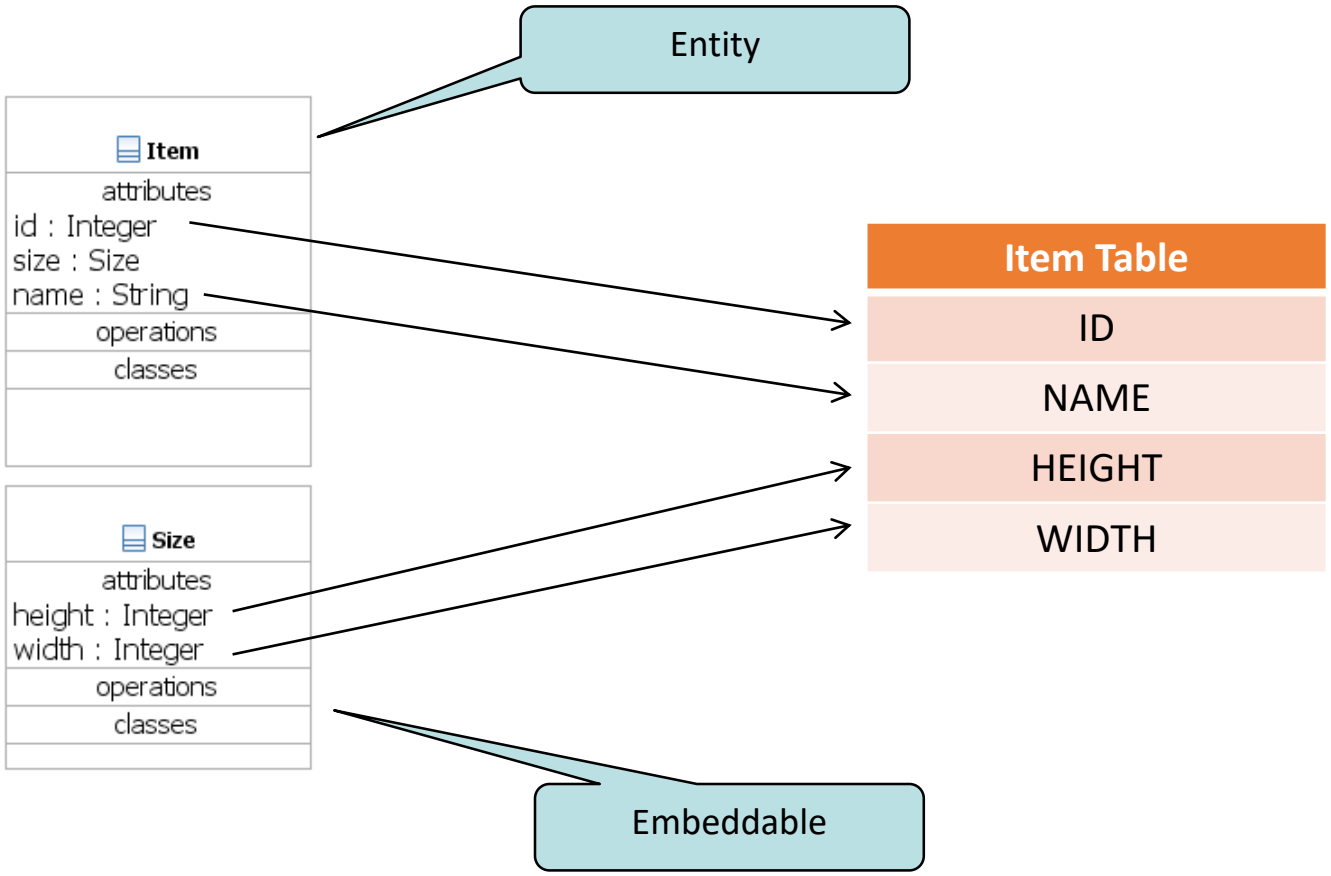
```
@Entity
@SequenceGenerator(name="CUSTOMER_SEQ", sequenceName="SEQ_T4471")
class Employee {

@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="CUSTOMER_SEQ")
@Column(name="CUSTOMER_ID")
public Long getId() { return id; }
```

Mapping Granularity

- Sometimes, the “class-per-table” approach is not sufficient.
- For example, we have the class *Size* that has width and height properties.
- The class Item contains a *Size* property.
- However, in the DB there is only one table: Item.
- See diagram on next page…

Mapping Granularity



Mapping Granularity

- As you can see, not every class is mapped to a separate table.
- Some classes should be **embedded** into others in the DB.
- In the previous example, the *Size* class is embedded into the *Item* class (from persistence point-of-view).

@Embeddable

- A class can be annotated with the **@Embeddable** annotation.
- This annotation specifies that the class is **not** mapped to its own table, but into its container’s table.
- Of course, the embeddable object can be used in different classes and thus be mapped into different tables.

Example

```
@Embeddable             // marking the class as Embeddable
public class Size {
        private int width;
        private int height;

        public Size() {      // required no-args constructor

        }
        public Size( int width, int height) {
                this.height = height;
                this.width = width;
        }
        public int getWidth() {return width;}
        public void setWidth(int width) {this.width = width;}
        public int getHeight() {return height;}
        public void setHeight(int height) {this.height = height;}
}
```

@Embedded

```
@Entity
public class Item {
    private String name;
    private int id;
    private Size size;

    public Item() {}
    public Item(String name) {this.name = name;}

    @Id
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    @Embedded
    public Size getSize() {return size;}
    public void setSize(Size size) {this.size = size;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
}
```

@AttributeOverride

- When you embed an *Embeddable* class, you can control the mapping of its properties.
- You do so by using the ***@AttributeOverrides*** annotation.

- Lets see an example…

Example

```
@Embeddable   // We changed this class property names
public class Size {
    private int w;
    private int h;

    public Size() {}

    public Size(int w, int h) {
        this.h = h;
        this.w = w;
    }

    public int getW() { return w;}

    public void setW(int w) {this.w = w;}

    public int getH() {return h;}

    public void setH(int h) {this.h = h;}
}
```

Example

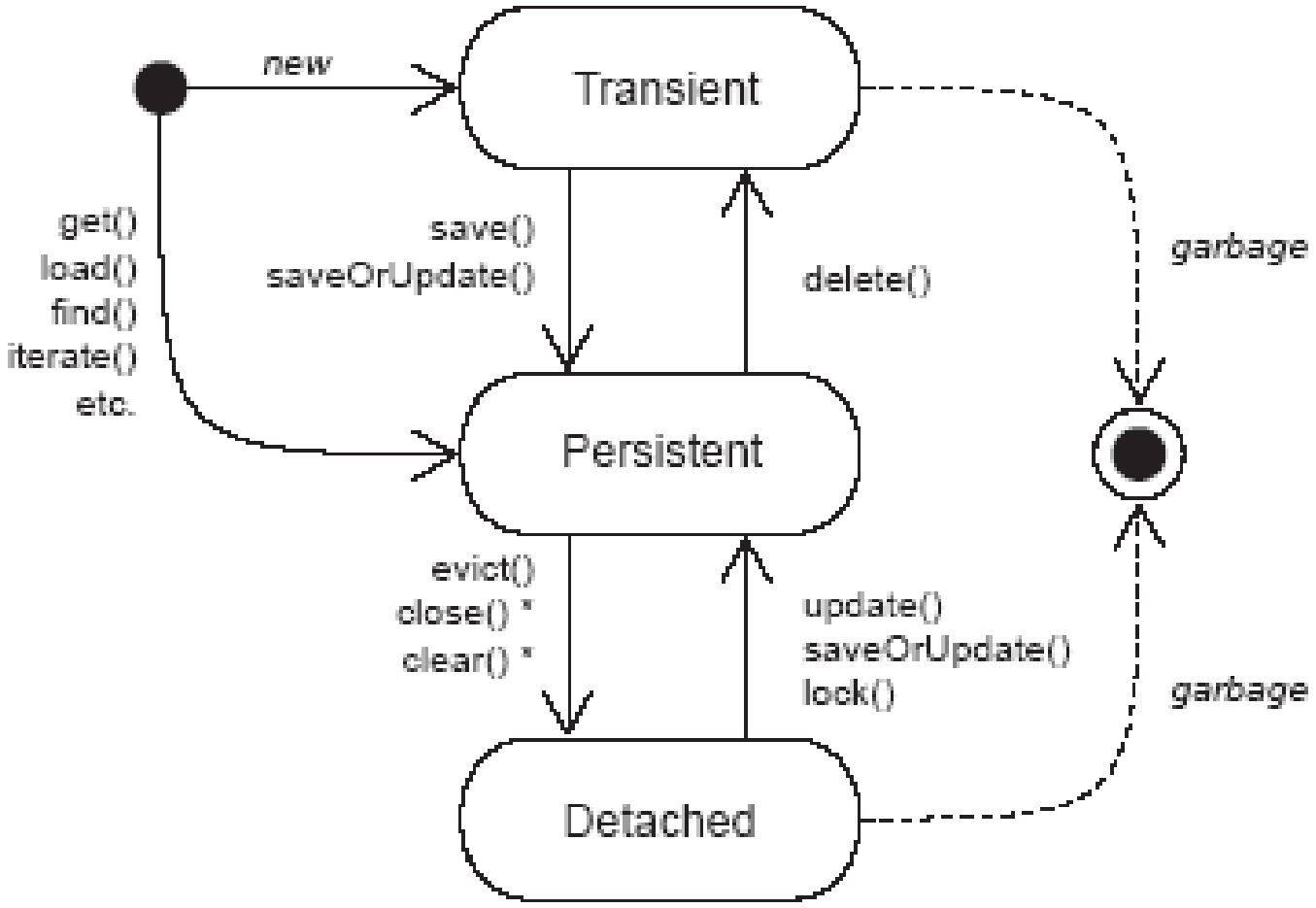
```
@Embedded
@AttributeOverrides( {
    @AttributeOverride(name = "w", column = @Column(name = "WIDTH")),
    @AttributeOverride(name = "h", column = @Column(name = "HEIGHT"))
})
public Size getSize() {
        return size;
}
```

Main player in ORM

@PersistenceContext – to inject EntityManager

- Session – hibernate interface
  - save
  - update
  - saveOrUpdate
  - delete
- EntityManager – JPA interface
  - persist
  - merge
  - remove

Persistent Object Lifecycle



Creating Queries

- Retrieving entities from the DB is done by using JPQL queries.
- The *EntityManager* has three methods for creating queries:
  - *createQuery()* – creates a regular query object.
  - *createNamedQuery()* – creates a named query.
  - *createNativeQuery()* – creates a regular SQL query.

Query Object

- The *Query* object supports method chaining (most methods return the *this*).
- The *Query* object supports paging via the *setFirstResult()* & *setMaxResults()* methods.

- See an example next page…

Example

- Selecting the 11-20 customers whose age is over 30:

```
int age = …
Query q = em.createQuery(
          "SELECT c FROM CUSTOMER c WHERE c.age > :age")
          .setParameter("age", age)
          .setFirstResult(11)
          .setMaxResults(10);
```

Transaction Management

Overview

- A transaction is a set of operations that should be executed in an *atomic* manner.
- I.e., either all the operations succeed, or all fail.
- A ‘good’ transaction has *ACID* properties.
- ACID → **A**tomic, **C**onsistent, **I**solated and **D**urable.

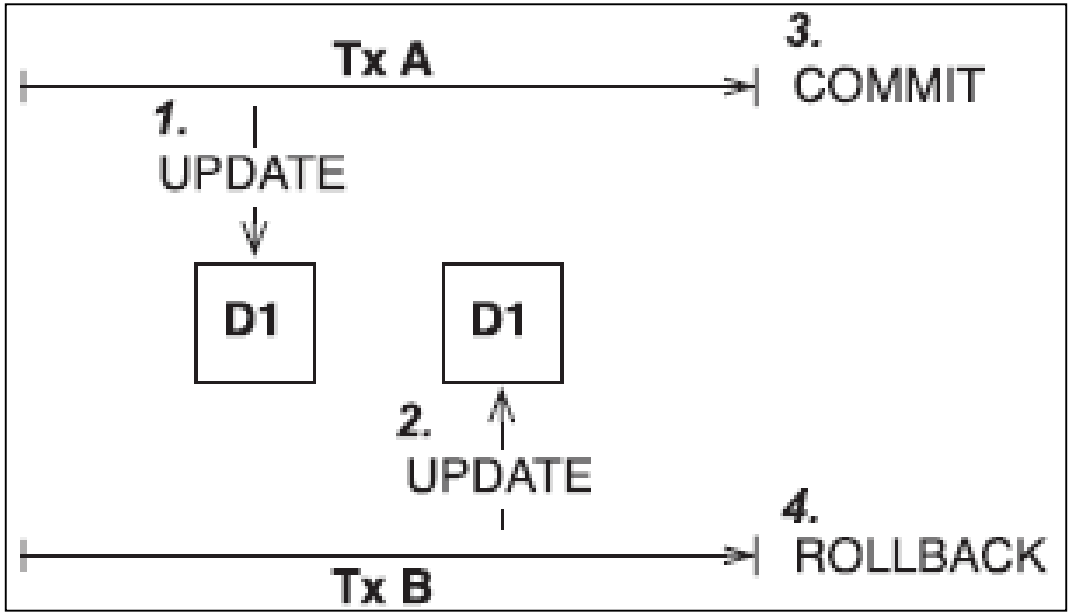
- Let’s see what it means…

The Isolated Property

- Lost Update - two transactions update the same date without locking, if one aborts, both changes lost
- Dirty Reads – one transaction may see uncommitted data from another transaction.
- Non-Repeatable Reads – one transaction may change data that another transaction currently uses.
- Phantom Reads – one transaction may insert a new data into a table that another transaction uses.

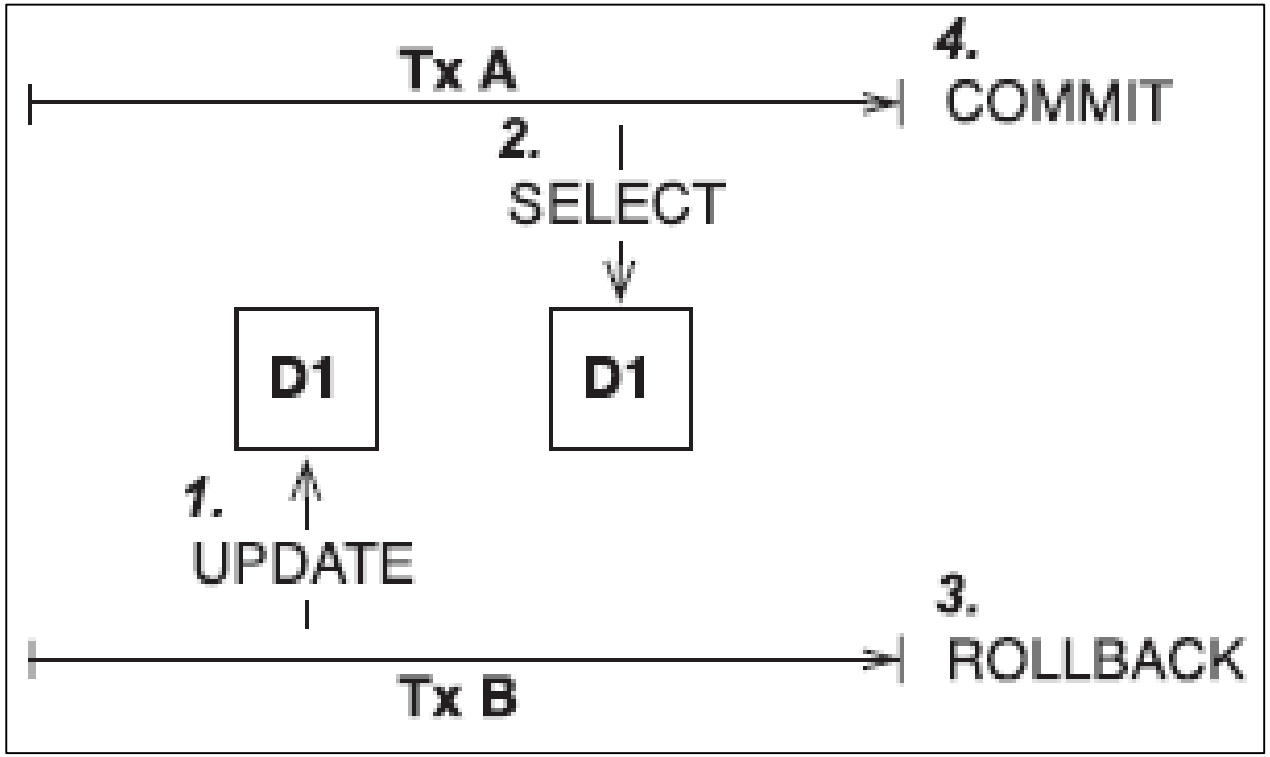
The Isolated Property

- Lost update
  - Two transactions update the same data without locking
  - If one of them aborts, both changes lost



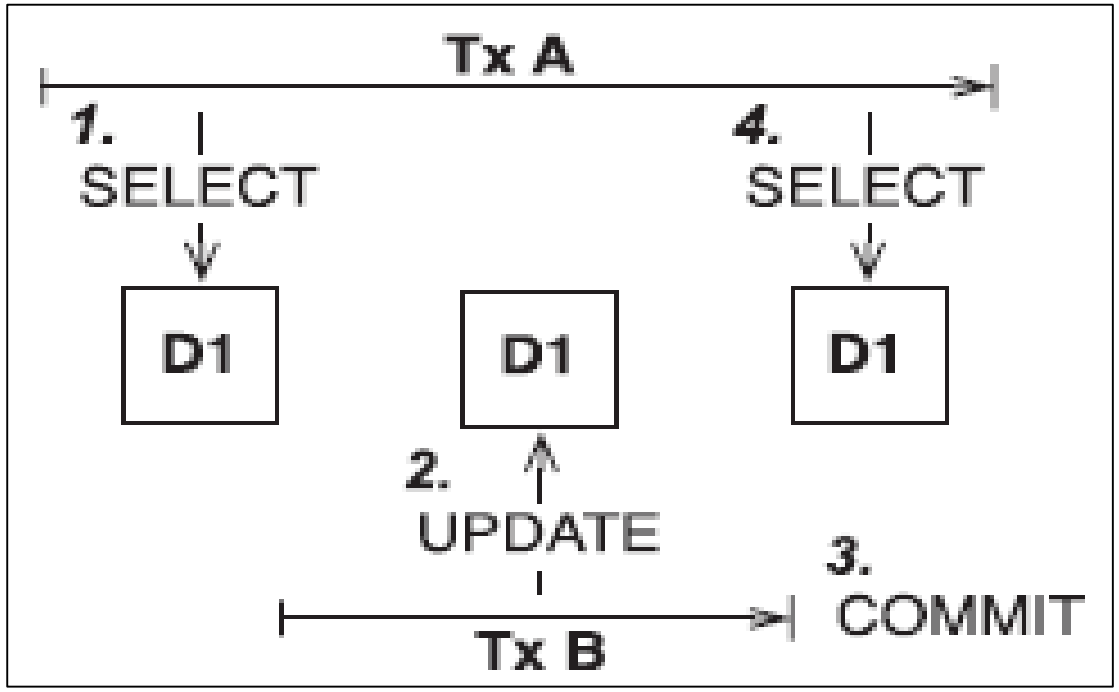
The Isolated Property

- Dirty read
  - Transaction A reads uncommitted data
  - The read data may never be committed



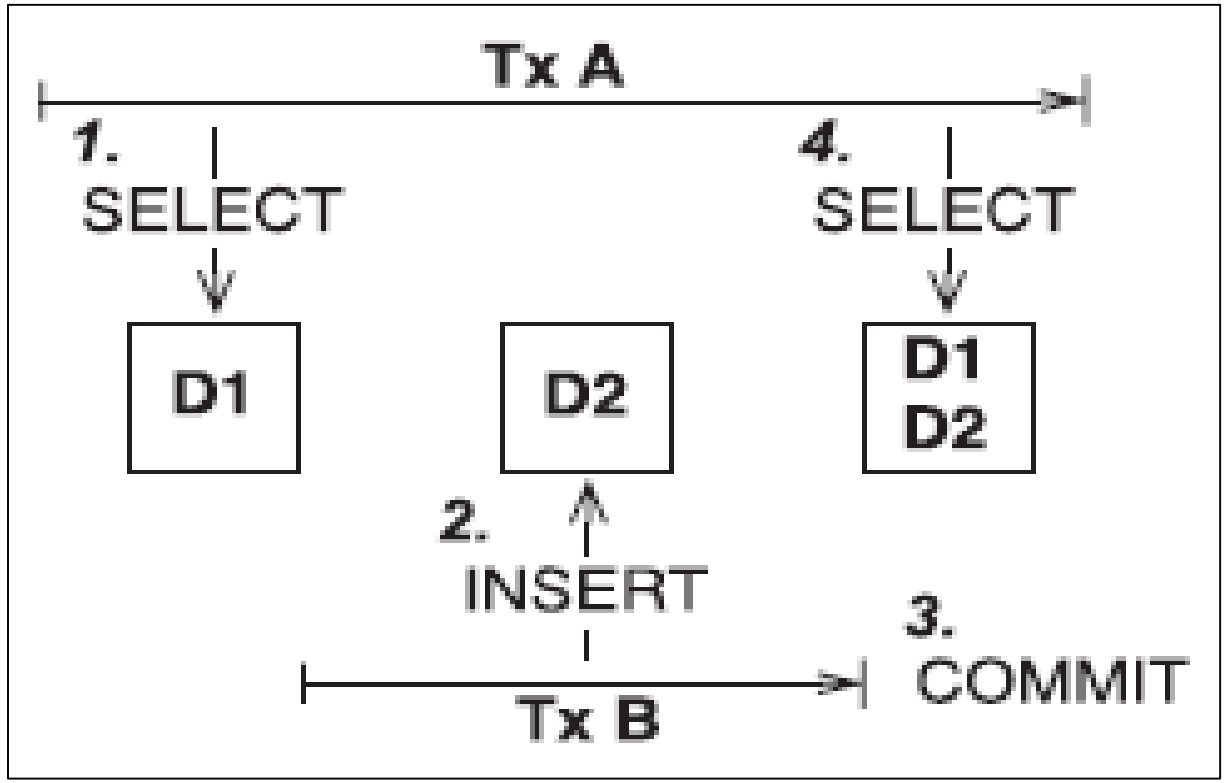
The Isolated Property

- Unrepeatable read
  - Transaction A executes two non-repeatable reads
  - If B updates between the reads



The Isolated Property

- Phantom read
  - Transaction A reads new data or lost data in the second select
  - If B inserts or deletes between reads



Transaction Types

- Two types of transactions exist:
  - Local transactions.
  - Global transactions.
- When working with an application server, we usually use **global** transactions.

Local Transactions

- A local transaction uses only **one** transactional resource (e.g., a single DB).
- Local transactions are **not** managed by the Application Server.
- They are under the responsibility of the developer.
- In order to use local transactions you should use the transactional resource-specific API.
- E.g., when working with JDBC, use – *Connection.setAutoCommit(**false**).*

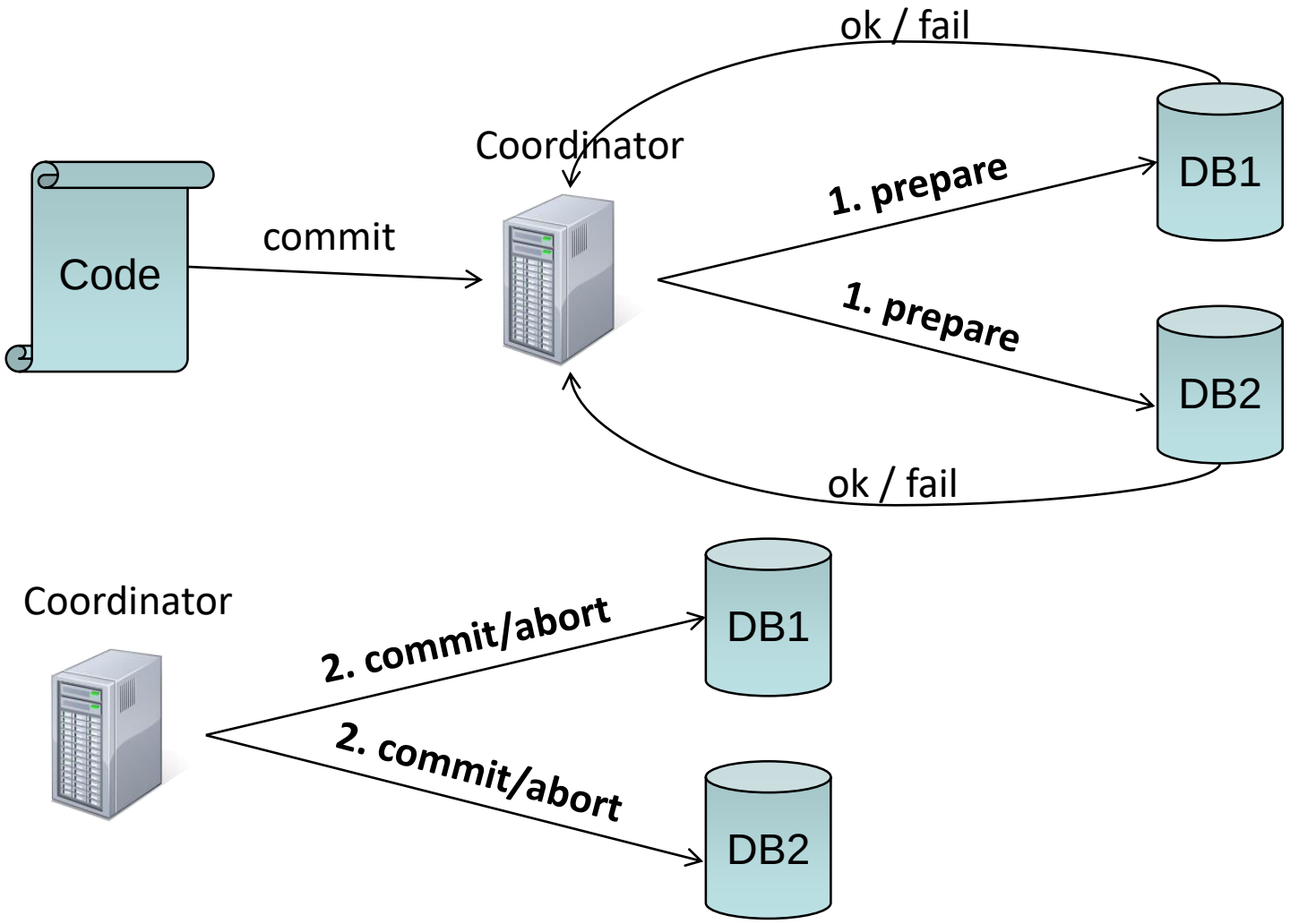
Global Transactions

- A global transaction may use more than one transactional resource.
- E.g., transferring money between accounts that reside on different DBs.
- Global transactions are also known as distributed transactions.
- And, as we are going to see, global transactions are complicated beasts…

The Problem with Global Txs

- If we try to perform a global transaction that involves two DBs using the JDBC API, we have a problem.
- Consider the following scenario:
  - Money was transferred between two accounts that reside on different DBs.
  - The transaction against the first DB has been committed.
  - And then, the second commit fails!

2 Phase Commit



Transactions & Spring

- Spring has a sophisticated transaction management.
- The key benefit that Spring provides is a complete abstraction of the underlying transaction mechanism.
- Hence, you need not change your code if you want to switch between local or global transaction management.
- Spring’s transaction management is considered to be very powerful and one of the best features that Spring offers.

PlatformTransactionManager

- At the center of Spring’s transaction management stands an implementation of the *PlatformTransactionManager* interface.
- Spring provides out-of-box implementations for JPA, JTA, Toplink, JDO, JMS, Hibernate, DataSource and JCA.
- A change in the transaction management strategy typically involves only changing the implementation class in the configuration file.

PlatformTransactionManager

- The *PlatformTransactionManager* is defined as follows:

```
public interface PlatformTransactionManager {
    // Returns a currently active transaction or
    // create a new one.
    TransactionStatus getTransaction(TransactionDefinition
        definition) throws TransactionException;

    void commit(TransactionStatus status) throws
        TransactionException;

    void rollback(TransactionStatus status) throws
        TransactionException;

}
```

The Benefits

- The *PlatformTransactionManager* approach has the following benefits:
  - It can be easily stubbed/mocked. http://martinfowler.com/articles/mocksArentStubs.html
  - Its methods declare throwing *TransactionException* which is an **unchecked** exception type.
  - It is not tied to the JNDI (like JTA in an application server).

Programmatic Transactions

- Spring offers two ways of using transactions programmatically:
  - Using the *TransactionTemplate* object.
  - Using the aforementioned *PlatformTransactionManager*.
- It is preferred to use the *TransactionTemplate* as it takes some of the burden of managing the transactions.

```
    public void updateCourses(final int newPrice) {
         template.execute(new TransactionCallback() {
         @Override
         public Object doInTransaction(TransactionStatus
                                  status) {
            Collection<Course> courses = dao.getCourses();
            for (Course course : courses) {
                 dao.updateCourse(course, newPrice);
            }
            // for void methods prefer using the
            // TransactionCallbackWithoutResult interface.
            return null;
         }
         });
    }
}
```

Summary

- The *TransactionTemplate* takes care of opening and closing the transaction.
- It is also responsible for exception handling.
- Rolling back the transaction is done by the *setRollbackOnly()* method of the *TransactionStatus* class.

More Configuration

- *TransactionTemplate* can also be configured with timeout, isolation level, propagation mode and read-only characteristics.
- Note that *TransactionTemplate* objects are thread-safe and stateless and can be used by many services simultaneously.

Declarative Transactions

- Spring provides declarative transactions using its AOP support.
- This makes sense as transactions are a cross-cutting concern.
- You specify the transaction behavior either in the XML configuration file or by using annotations.

Rollbacks

- If there are no exceptions and you don’t programmatically set the *rollback-only* flag, the transaction will be committed.
- By default, Spring will perform a rollback only for **Runtime Exceptions** or **Errors**.
- Hence, checked exceptions will **not** trigger a rollback!
- This, of course, can be configured.

Configuring Attributes

- The transaction's attributes can be configured in the *<tx:advice>* tag.
- We have already seen the '*name*', '*rollback-for*' and '*no-rollback-for*' attributes.
- More configuration attributes are available:
  - propagation – defaults to REQUIRED.
  - isolation – sets the isolation level.
  - timeout – timeout in seconds (defaults to -1).
  - read-only – defaults to *false*.

Propagation Behavior

- Spring’s transactions mechanism supports the following propagation behaviors (most are analogous to the EJB standard):
  - REQUIRED - (default)
  - REQUIRES_NEW
  - MANDATORY
  - SUPPORTS
  - NOT_SUPPORTED
  - NEVER
  - **NESTED**

Propagation Value	Existing Transaction	Transaction Associated with Business Method
REQUIRED	None T1	T2 T1
REQUIRES_NEW	None T1	T2 T3
MANDATORY	None T1	Error T1
SUPPORTS	None T1	None T1
NOT_SUPPORTED	None T1	None None
NEVER	None T1	None Error

Transactions using Annotations

```
@Transactional(readOnly = true)
public class CourseServiceImpl implements CourseService {
   public void deleteCourse(Course c) {   }
   public Course getCourse() { return null; }

   @Transactional(readOnly = false, noRollbackFor =
                UnsupportedOperationException.class)
   public void insertCourse(Course c) {
        throw new UnsupportedOperationException();
   }
   public void updateCourse(Course c) {
        throw new UnsupportedOperationException();
   }
}
```

Activating Annotations

@EnableTransactionManagement

```
<tx:annotation-driven transaction-manager="txManager"/>
```

- ```
  <bean name="bookstore" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="url" value="jdbc:derby:bookstore;create=true"/>
  </bean>
  ```
- ```
  <bean id="bookstore-persistence-unit"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="bookstore"/>
      <property name="persistenceProviderClass" value="org.hibernate.ejb.HibernatePersistence"/>
      <property name="jpaVendorAdapter">
          <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
      </property>
      <property name="jpaProperties">
          <props>
              <prop key="hibernate.dialect">org.hibernate.dialect.DerbyTenSevenDialect</prop>
              <prop key="hibernate.hbm2ddl.auto">create</prop>
              <prop key="hibernate.show_sql">true</prop>
              <prop key="hibernate.format_sql">true</prop>
          </props>
      </property>
      <property name="packagesToScan">
          <list>
              <value>bookstore</value>
          </list>
      </property>
  </bean>
  ```
- ```
  <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="bookstore-persistence-unit"/>
  </bean>
  ```
- ```
  <context:component-scan base-package="bookstore"/>
  ```
- ```
  <tx:annotation-driven transaction-manager="txManager"/>
  ```
  ```