# SPRING BOOT: DEPENDENCY INJECTION, AOP & TRANSACTIONS

DAY 1

# COURSE OVERVIEW

# TODAY'S TOPICS

SPRING BEAN

APPLICATION CONTEXT

@COMPONENT

DEPENDENCIES AND DEPENDENCY INJECTION

ASPECT ORIENTED PROGRAMMING (AOP)

TRANSACTIONS

# HOW DOES SPRING DO ITS MAGIC?

- Adds enterprise services to POJOs
- Dependency injection and inversion of control (coming soon)
- Spring container does most of the magic

# EXERCISE

- Take 10 minutes in pairs to come up with a two sentence explanation of inversion of control and dependency injection.

# INVERSION OF CONTROL AND DEPENDENCY INJECTION

## Inversion of Control

Inversion of control: control flow not in hands of the developer but in the hands of the framework.

Dependency injection: one way of applying inversion of control. External code (framework) injects the dependencies to the class.

Old days:

The classes (and manager classes for each class) take care of instantiation and accessibility and connections etc their selves by calling on libraries.

Now:

The framework is in control and does all these things!

```
public class ExampleController {
    private ExampleService exampleService;

    ExampleController(){
        this.exampleService = new ExampleService();
    }
}
```

```
@Controller
public class ExampleController {
    private ExampleService exampleService;

    @Autowired
    ExampleController(ExampleService exampleService){
        this.exampleService = exampleService;
    }
}
```

# EXERCISE

TAKE 5 MINUTES TO COME UP WITH A TWO SENTENCE EXPLANATION OF SPRING BEANS.

# SPRING BEANS

**Bean is an instance of a class (object), that is instantiated and managed by the IoC Spring container.**

The framework is in control of which beans are instantiated and ended.

In code you can recognize beans like this: @..... (and then some Bean annotation like @Bean, @Component, @Controller, @Repository etc.)

Not all classes are beans, only the ones that need to be managed by the container. Examples of beans in everyday enterprise applications are REST controllers, models/entities and repositories.

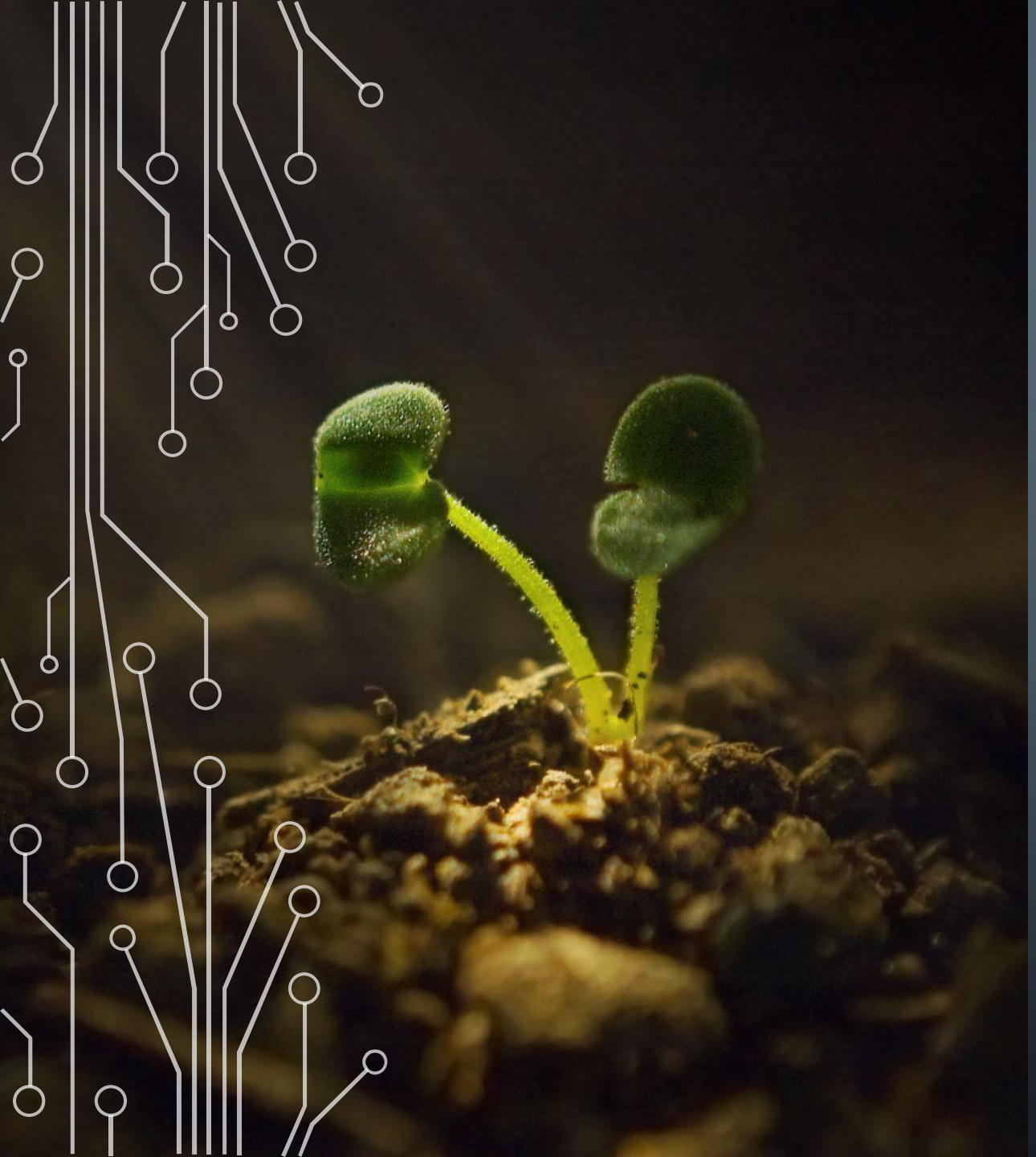
The container manages the lifecycle of the beans, the transactions, security and other system-level services.

# EXERCISE

- So: a bean is an object, that is instantiated by the IoC Spring container.

- But what is this Spring container?

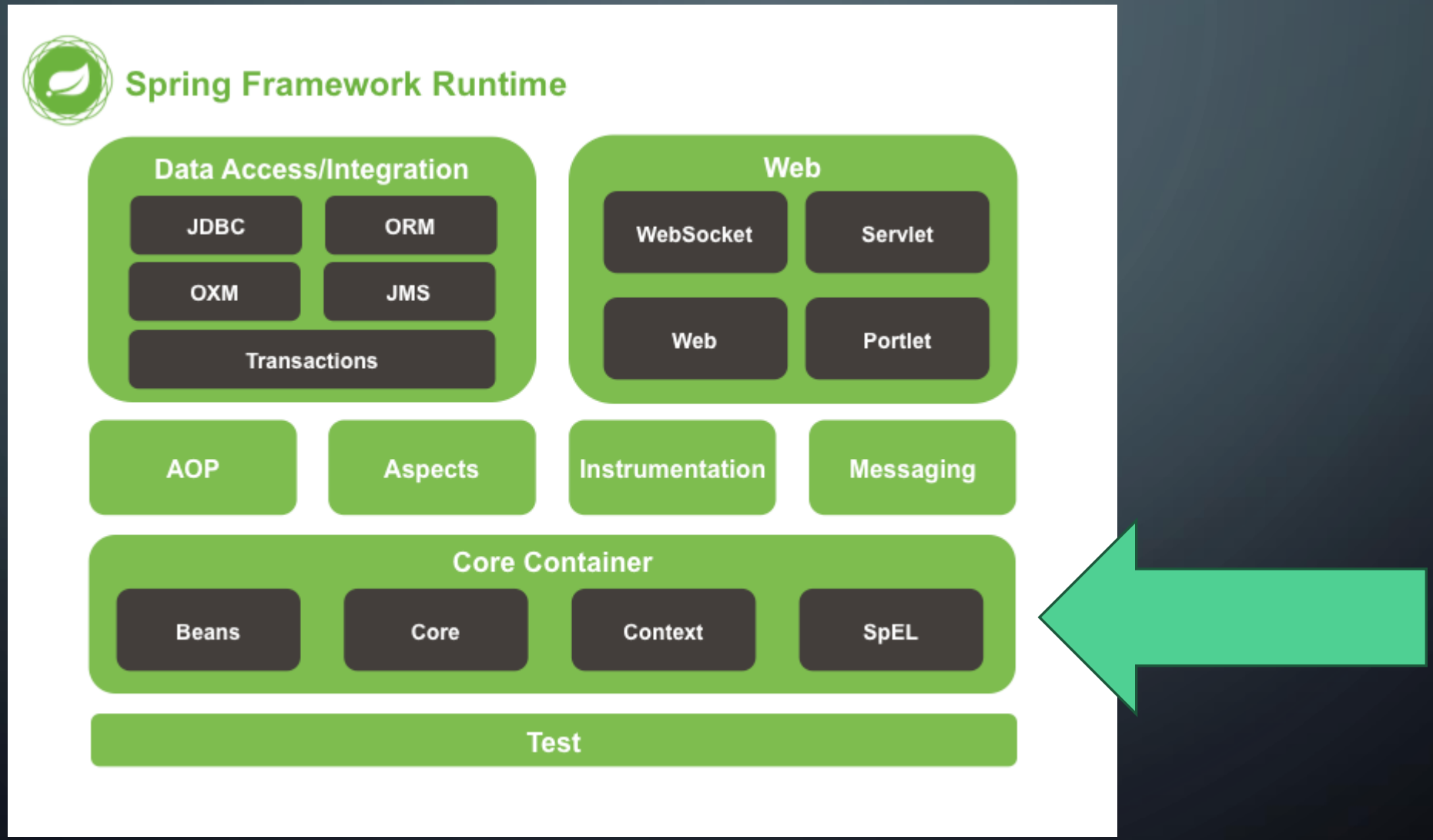
  Take 5 minutes to come up with a two sentence explanation of Spring container.

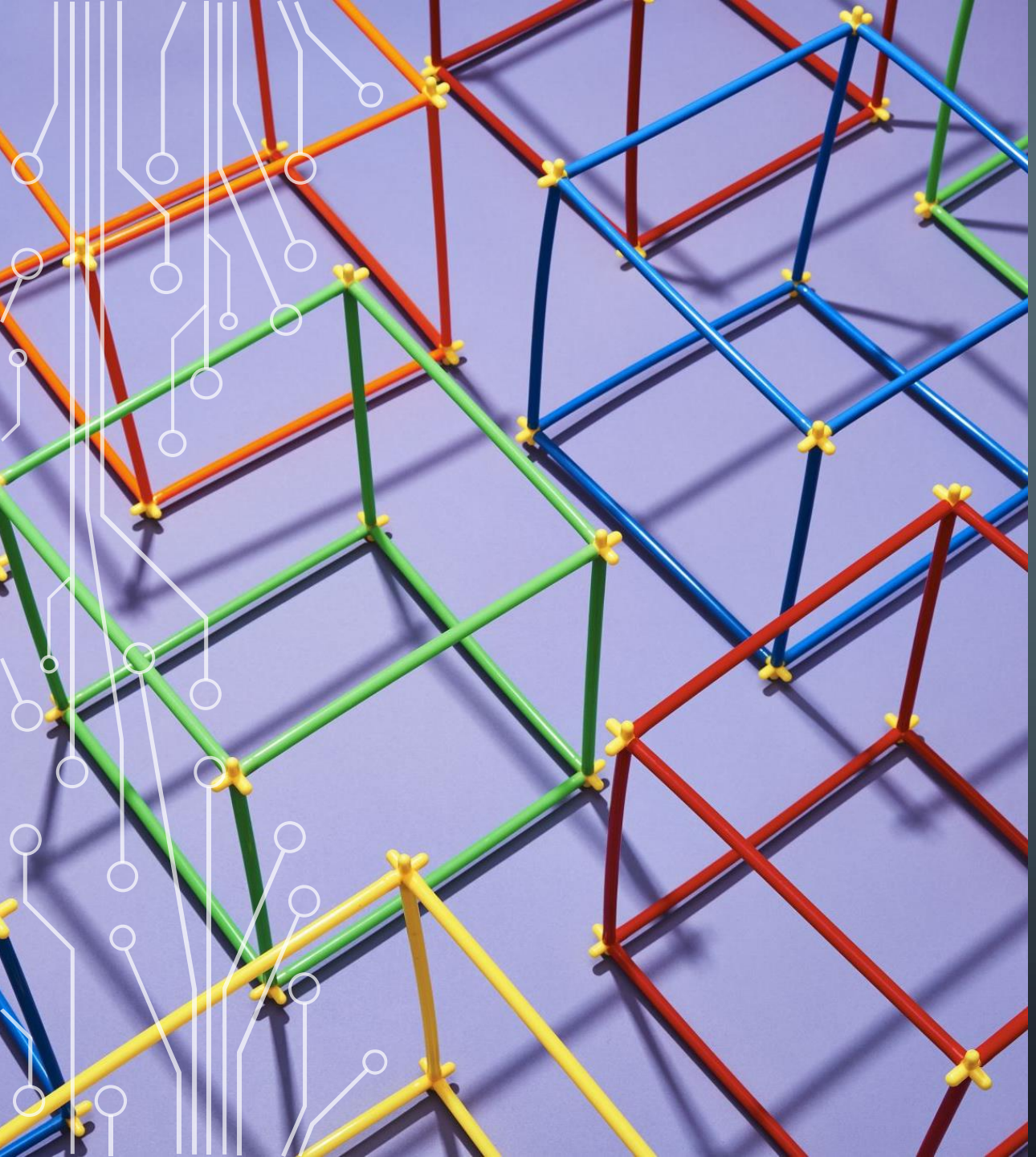


# SPRING CONTAINER

- Core of the Spring framework
- Responsible for the life cycle of the beans
- Creates the objects of the beans, wires them together and configures them, destroys them
- Uses Dependency Injection

# SPRING CONTAINER





# APPLICATION CONTEXT

- Part of Spring's container (together with BeanFactory (later))
- Registry that holds and wires bean definitions
- Add enterprise-specific functionality
- Different implementations available (it's an interface ApplicationContext)

# DEMO – BEANS AND DEPENDENCY INJECTION

# COMPONENT SCANNING IN SPRING

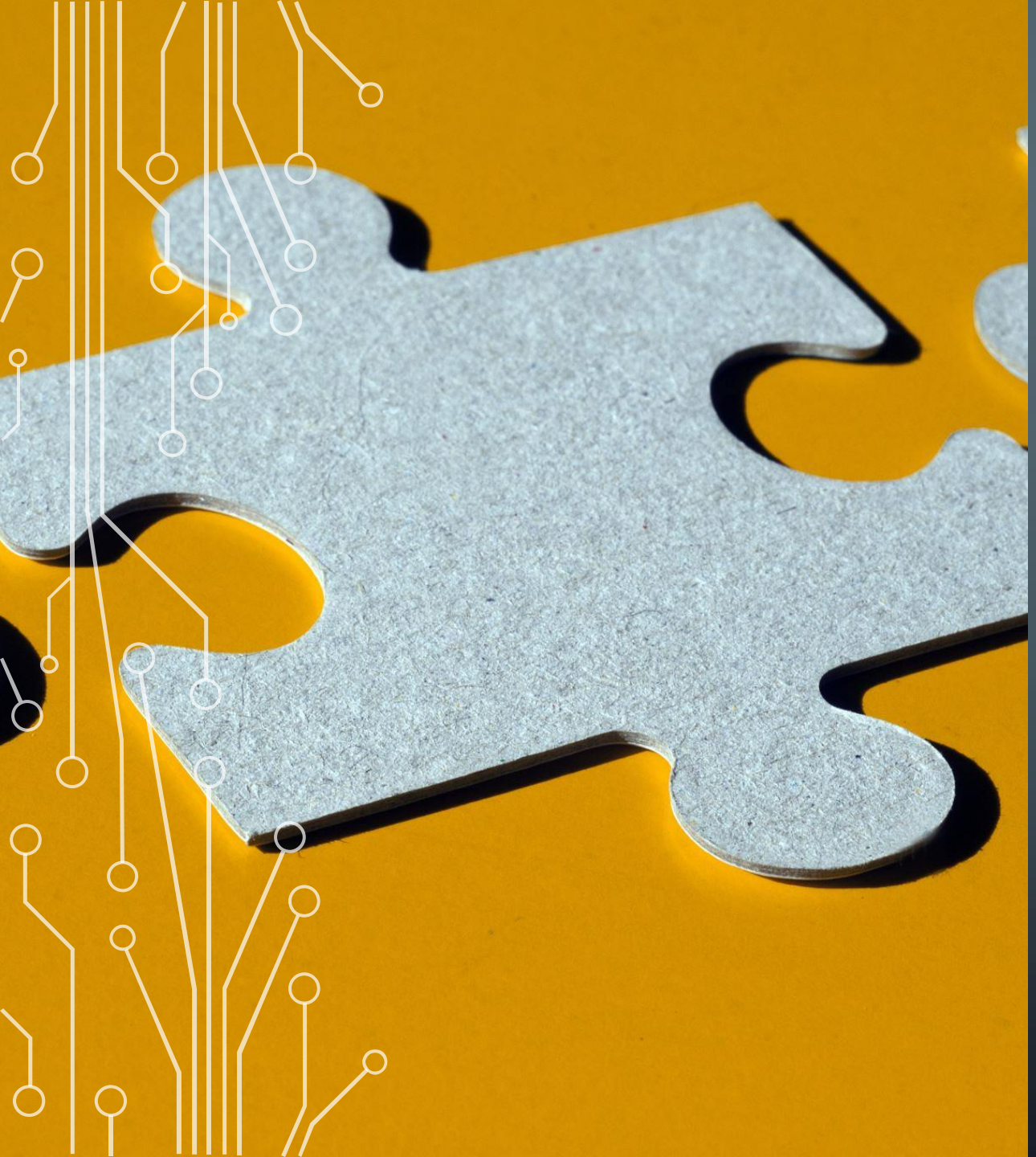
- Spring automatically scans for beans
- @ComponentScan scans the package of the class that annotions is on and all the subpackages
- If main method class is also the configuration class (so no custom @Configuration) the scanning will start from there
- If Spring needs to inject something that is not a bean, you'll get an exception

# DEMO – EXCEPTIONAL DEMO

LET'S DO THE COMPONENT SCANNING WRONG AND SEE WHAT HAPPENS.

# ASSIGNMENT 1 – DEPENDENCY INJECTION

- Create a new Spring app for this exercise.
- Define an interface PaymentService with a method processPayment.
- Implement this interface in three different classes: CreditCardPaymentService, PaypalPaymentService, and BankTransferPaymentService. Each implementation should have a simple print statement to identify which service is being processed.
- Configure Spring to inject these services into a single PaymentProcessor class based on different scenarios:
  - Use the @Primary annotation to default to one service.
  - Use the @Qualifier annotation to inject specific services into different instances of another bean.
  - Bonus: use Spring profiles to switch between implementations based on the active profile
- Test your application.



# EXERCISE

- We have used them already, but what are they…?
- Take 5 minutes to come up with a two sentence explanation of Spring annotations.

# XML

- In the olden days, all configuration happened via XML
- Application context stored in applicationContext.xml
- You won't find this file in new projects

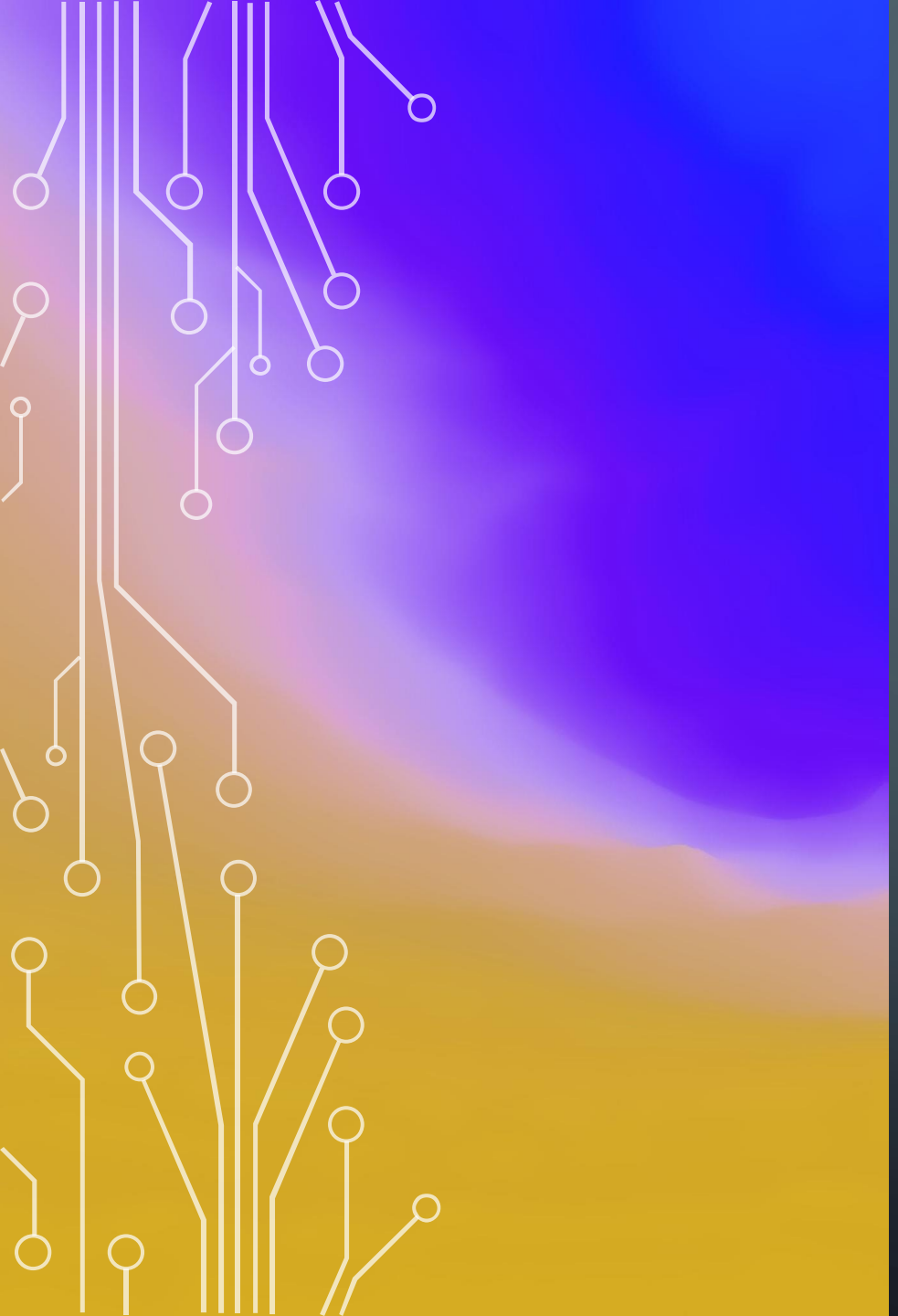
- Developers didn't like all this XML config and therefore they've made Java configuration possible.

# SPRING ANNOTATIONS

- You have all used them
- Before the beans and annotations would be configured in xml, more and more frameworks are moving to annotation based configurations.
- Luckily we now have annotations.

```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-3.0.xsd">

   <!-- A simple bean definition -->
   <bean id = "..." class = "...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- A bean definition with lazy init set on -->
   <bean id = "..." class = "..." lazy-init = "true">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- A bean definition with initialization method -->
   <bean id = "..." class = "..." init-method = "...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- A bean definition with destruction method -->
   <bean id = "..." class = "..." destroy-method = "...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- more bean definitions go here -->

</beans>
```

```java
@Configuration
public abstract class VisibilityConfiguration {

  @Bean
  public Bean publicBean() {
     Bean bean = new Bean();
     bean.setDependency(hiddenBean());
     return bean;
  }

  @Bean
  protected HiddenBean hiddenBean() {
     return new Bean("protected bean");
  }

  @Bean
  HiddenBean secretBean() {
     Bean bean = new Bean("package-private bean");
     // hidden beans can access beans defined in the 'owning' context
     bean.setDependency(outsideBean());
  }

  @ExternalBean
  public abstract Bean outsideBean()
}
```



# SPRING ANNOTATIONS

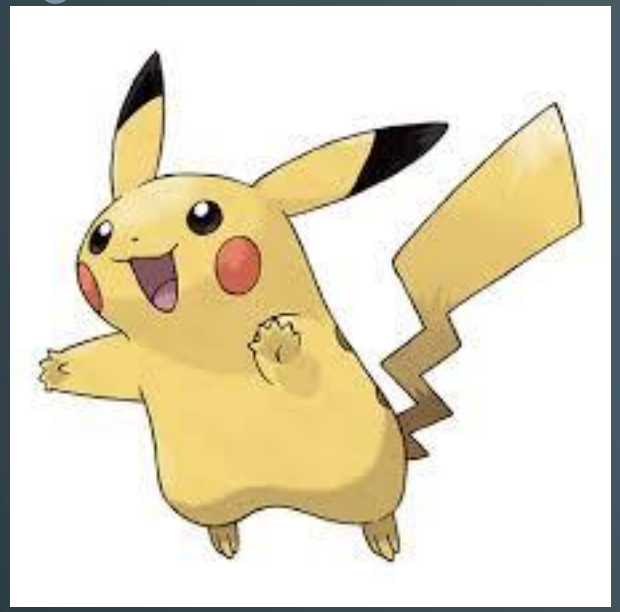
- Way to easily configure behavior of a piece of code
- Several kinds of annotations:
  - Core spring framework annotations
  - Stereotype annotations
  - Spring Boot annotations
  - REST and MVC annotations
  - Mapping annotations
  - Spring Cloud annotations
  - And more…
- You can even make custom annotations!

# ANNOTATIONS

```java
package com.helloworld.helloworld.service;

import ...

@Service
public class HelloworldService {

    @Autowired
    HelloworldRepository<Helloworld> helloworldRepository;

    public List<Helloworld> helloworld() { return helloworldRepository.findAll(); }
}
```

```java
package com.helloworld.helloworld.controller;

import ...

@RestController
public class HelloworldController {

    @Autowired
    HelloworldService helloworldService;

    @RequestMapping(value = "/helloworld", method = RequestMethod.GET)
    public String helloworld() {
        //return helloworldService.helloworld();
        return "helloworld";
    }
}
```

```java
@Entity
@Table(name="helloworld")
public class Helloworld {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name="helloworld")
    private String helloworld;

    public Helloworld() {}
```

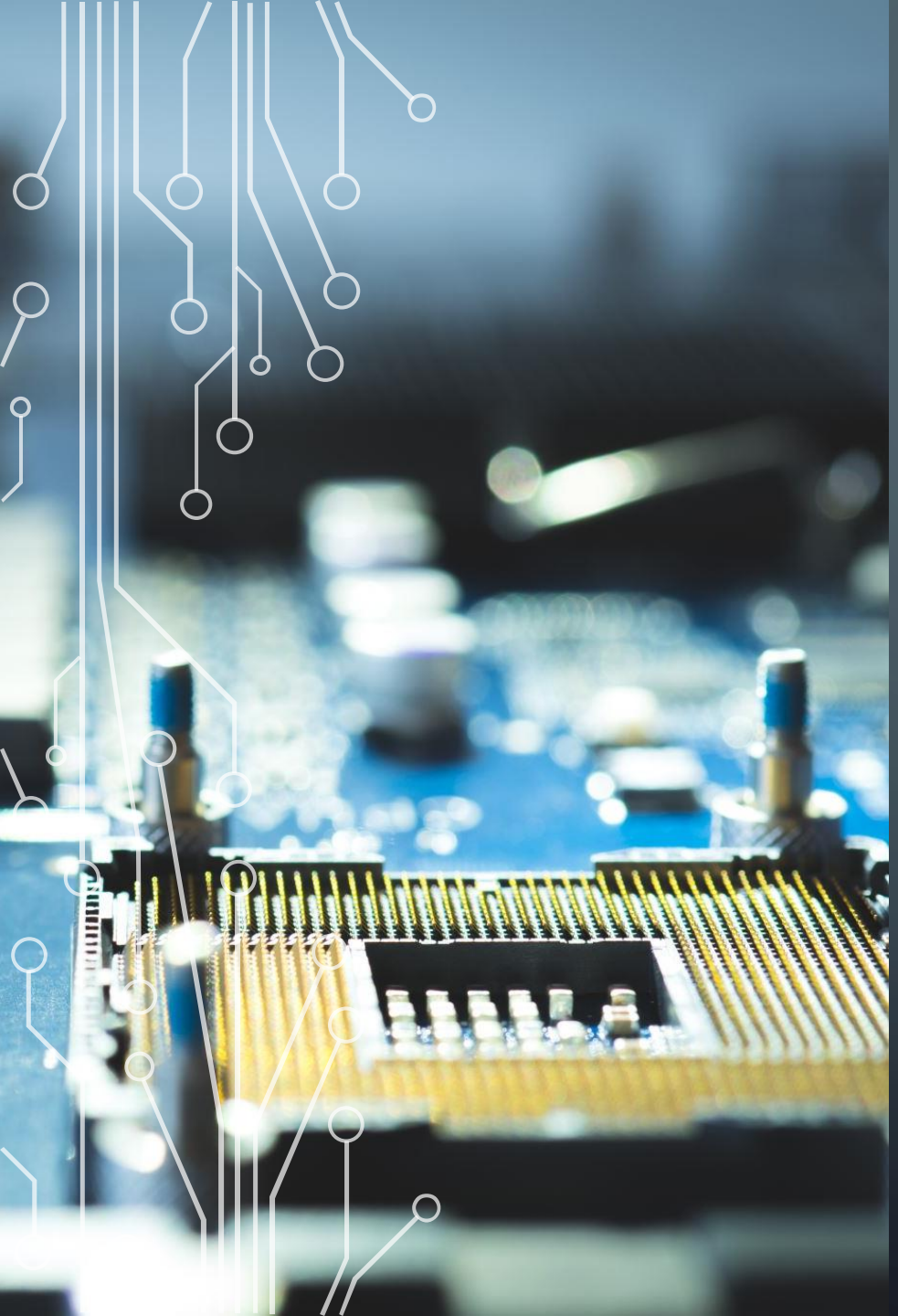


- Four of these annotations don't belong in Spring and one these four is a Pokemon, do you know which ones?

- **Annotations**
  - @Repository
  - @Singleton
  - @Magneton
  - @Multiton
  - @Controller

- **Annotations**
  - @PathVariable
  - @ModelAttribute
  - @Component
  - @Service
  - @RestController
  - @SecureController

- **Annotations**
  - @Autowired
  - @RequiredArgsConstructor
  - @Auto
  - @SpringBootApplication
  - @EnableAutoConfiguration
  - @ComponentScan
  - @Configuration
  - @EnableTransactionManagement
  - @EnableJpaRepositories
  - @PropertySource
  - @Entity



# JAVA CONFIGURATION - @CONFIGURATION

- Adding a Java class for config (e.g. called AppConfig.java)
- This class has the annotation @Configuration
- Replacement for the old XML config files

# QUESTION: IS THE CONFIGURATION CLASS A BEAN?

# QUESTION: IS THE CONFIGURATION CLASS A BEAN?

- YES!

# @BEAN

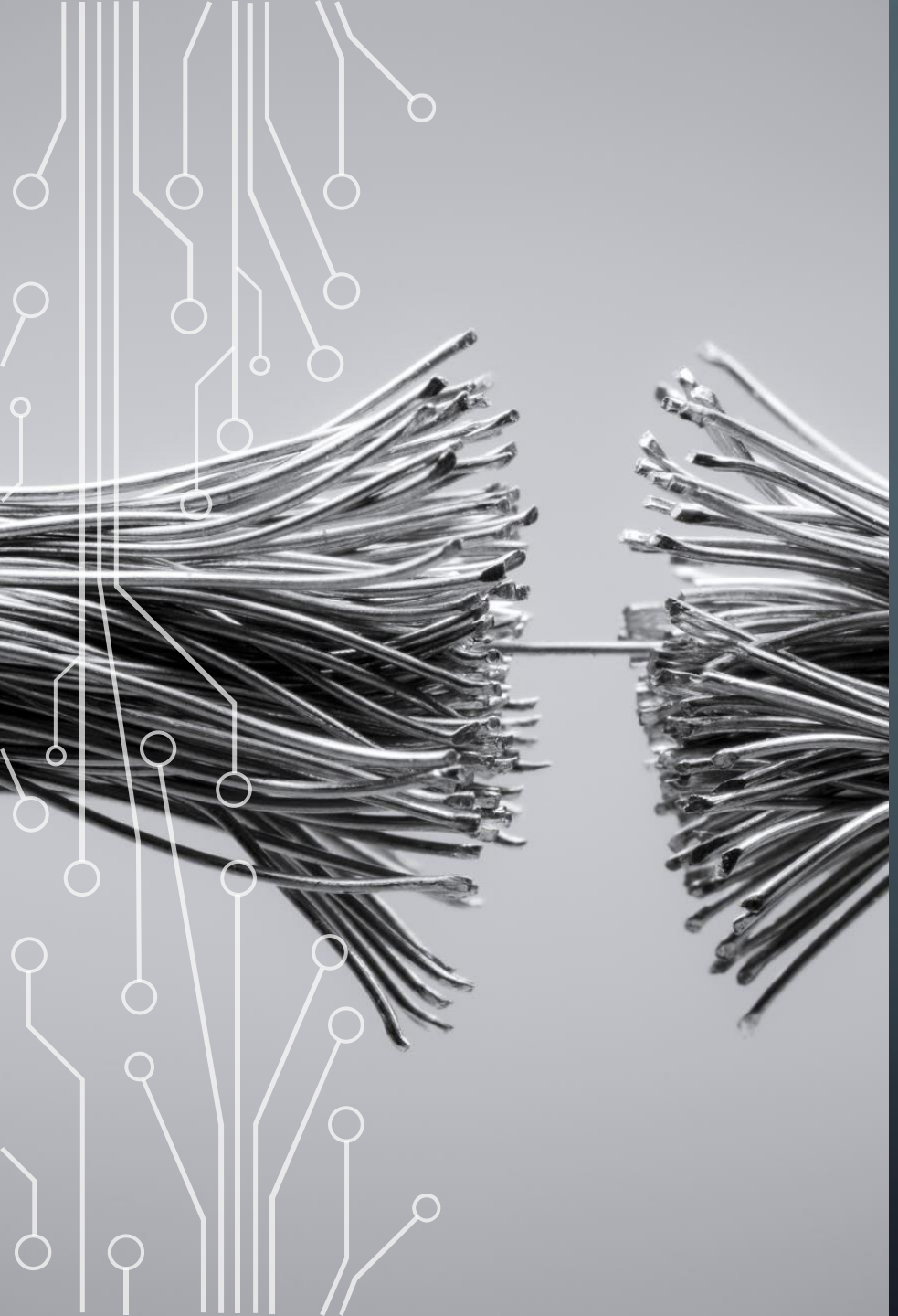
- Add @Bean to a method in the configuration class
- This enables injection: setter, constructor, field
- Default all @Bean are singleton and only created the first time the method gets called

# CONSTRUCTOR INJECTION

- @Autowired on top of constructor
- Dependencies are provided to the bean through the constructor
- When there's only one constructor that takes arguments, dependency injection will happen automagically. Still a best practice to add @Autowired on top.

# SETTER INJECTION

- Injection through the setter, autowired is on top of a setter
- Useful if you don't have a constructor
- Allows reconfiguration later, new bean can be set as a dependency
- Third party code only supports setter injection



# FIELD INJECTION

- Autowired on top of the field
- Recommended to not use this too much, because it's heavy on the performance since it works with reflection API beneath the surface

# EXERCISE

- Can you come up with use cases for field injection?
- Show a code example

# CAN YOU COME UP WITH USE CASES FOR FIELD INJECTION?

- When you have a @Configuration class with a certain bean, that depends on a bean of another @Configuration class
- Inject beans of the Spring container infrastructure that you'd need to modify (which is terrible, because you tie your code to the framework, but still this is a use case)
- Tested bean in test classes, keeps it readable and performance less relevant


# EXERCISE

- Some extra topics:
  - @Qualifier
  - @Value
  - Required property on @Autowired
  - @AliasFor
  - @Lazy
- Tell me in groups of 3:
  - What is it?
  - Demonstrate with a little piece of code
  - What are the pros/cons for using it?

# HOW DOES SPRING DECIDE WHAT TO INJECT?

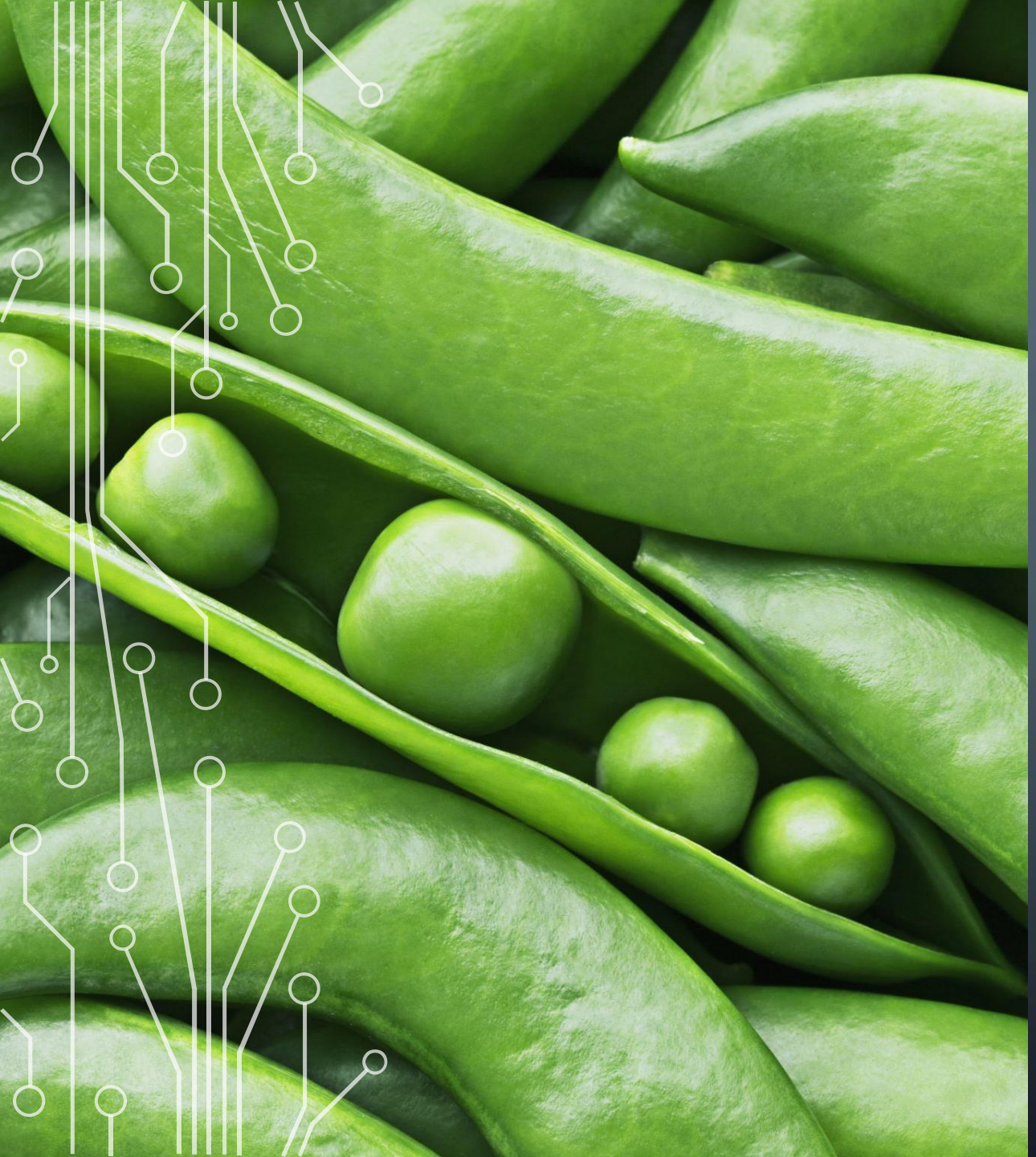
First on type: this works if there's only one bean of that type

If there is more than one, it is looking for the bean with @Qualifier annotation. After that, @Primary.

If nothing is found, it tries to autowire by name. Default name is class name/method name starting with lowercase letter

# WHAT EXCEPTION DO YOU GET IF SPRING CANNOT FIND A BEAN TO INJECT?

AND WHAT IF IT'S AMBIGUE?



# WHAT EXCEPTION DO YOU GET IF SPRING CANNOT FIND A BEAN TO INJECT?

- NoSuchBeanException
- UnsatisfiedDependencyException - NoUniqueBeanDefinitionException

# BEAN SCOPES @SCOPE

- Singleton (default scope, valid in all configs) – one instance per Spring container
- Prototype (valid in all configs) – one instance per request
- Request (valid in web-aware Spring projects) – one instance per lifecycle
- Session (valid in web-aware Spring projects) – one instance per user session
- Global (valid in web-aware Spring projects) – one instance per global session (life of application on server, so new one after reboot)

- So e.g. @Scope(value="session") or @Scope(value=BeanDefinition.SCOPE_SESSION)

# EXERCISE: SCOPES

- Create a Spring Boot application that illustrates the behavior of Singleton and Prototype scoped beans through a simple service that manages user sessions and requests:
  - Service:
    - Implement a service named SessionManagerService that will simulate managing user sessions.
    - Implement another service named RequestProcessorService that will simulate processing user requests.

# EXERCISE: SCOPES 'CONT'D

- Scope Definition:
  - Annotate SessionManagerService with @Scope("singleton") to indicate that it should be a Singleton scoped bean.
  - Annotate RequestProcessorService with @Scope("prototype") to indicate that it should be a Prototype scoped bean.
- Functionality:
  - In SessionManagerService, implement a method that returns the instance's hash code (to identify the instance).
  - In RequestProcessorService, implement a similar method that returns the instance's hash code.
- Controller Implementation:
  - Create a TestController with two endpoints: /session and /request.
  - Inject both SessionManagerService and RequestProcessorService into this controller.
  - For /session, call the method in SessionManagerService that returns the instance's hash code and return it to the client.
  - For /request, do the same with RequestProcessorService.
- Hit both endpoints multiple times and observe the difference.

# CHALLENGE

- What if a prototype scoped bean is injected into a singleton scoped bean?

# CHALLENGE

- What if a prototype scoped bean is injected into a singleton scoped bean?

- Singleton: A singleton scoped bean is instantiated once per Spring IoC container. Every time you request the bean, Spring provides the same instance.
- Prototype: A prototype scoped bean is created anew each time it is requested from the container.
- So, when a prototype scoped bean is injected into a singleton scoped bean, the prototype bean is instantiated and injected at the time the singleton bean is created.
- Despite being prototype scoped, within the context of the singleton bean, it behaves like a singleton because it's only requested once. This means the supposed prototype bean does not get recreated on each call or request within the singleton; it remains the same instance that was initially injected when the singleton bean was created.

# CHALLENGE

- To preserve the prototype behavior of a bean when it's used within a singleton scoped bean, Spring provides the @Lookup annotation

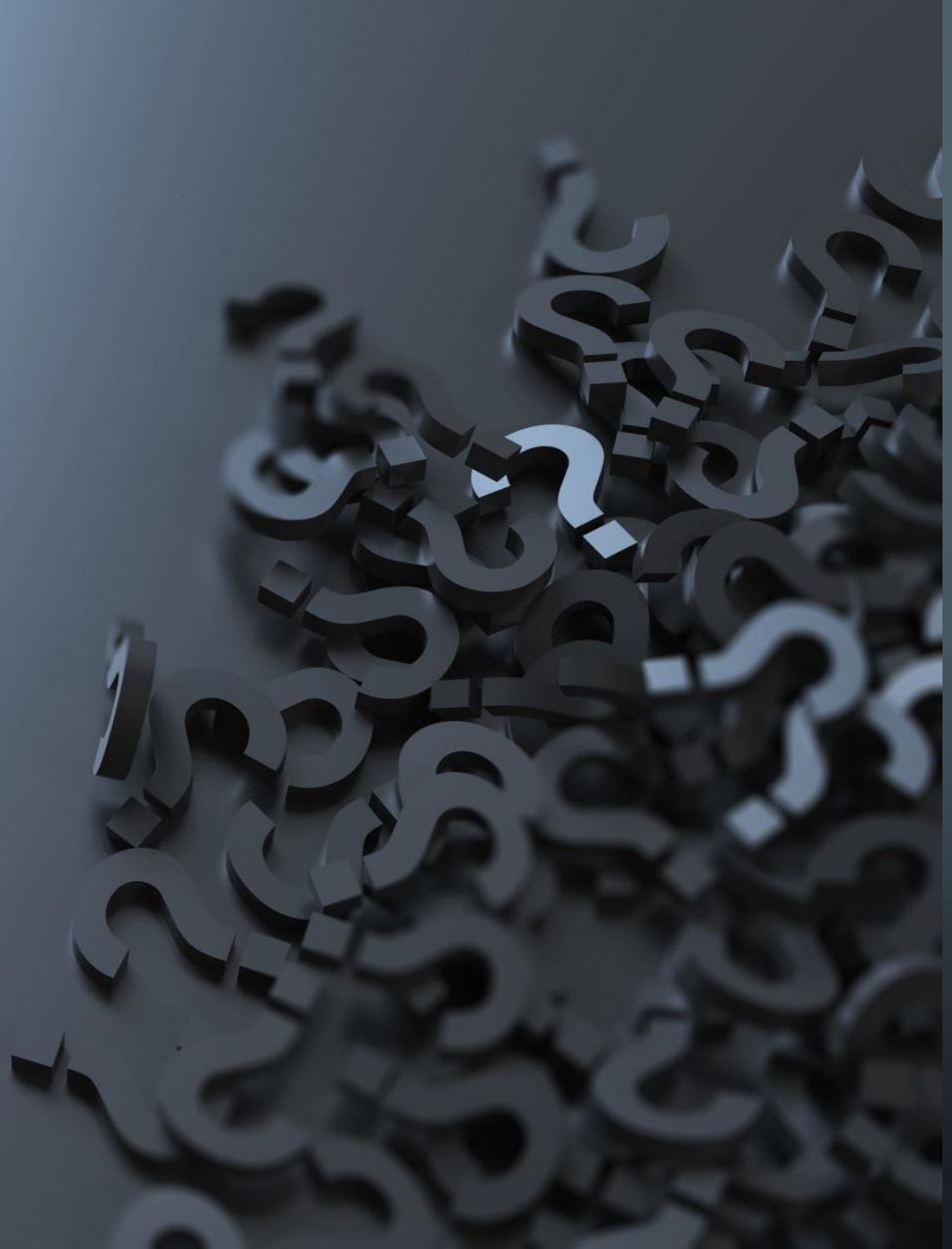
- By annotating a method in the singleton bean with @Lookup, Spring dynamically overrides the method to return the bean from the application context directly, effectively returning a new instance each time for a prototype bean.

# QUESTION

- Is it better to create a bean using @Bean or @Component (or one of the stereotype annotations)?
- When should you be using each?

# LIFECYCLE OF A BEAN

- Instantiation
- Populate properties
- BeanNameAware
- BeanFactoryAware
- BeanPreProcessors (pre initialize)
- InitializeBean
- Init
- BeanPostProcessors (post initialize) (@PostConstruct)



# QUESTION

- Take 10 minutes to come up with a simple explanation for Bean Factory

# BEAN FACTORY

Pattern to encapsulate object construction logic in a class.

It is a done in a way that the construction can be reused.

- Expands initMethod concept
- Factory method pattern
- Deal with legacy code
- Contract with constructor
- Static methods

https://www.youtube.com/watch?v=xIWwMSu5I70

# MULTIPLE CONFIGURATION CLASSES

- You can import another configuration class to a configuration class with the @Import annotation on top of the config class

- @Import(MyOtherConfig.class)

- Exercise: Demonstrate this, by using a bean from another config file that gets imported by @Import

# ASPECT ORIENTED PROGRAMMING (AOP)

Theory

- Way to add extra behavior to your code without changing the code itself.

- AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.

- These concerns include logging, security checks, or transaction management—things that you usually need across different parts of your application.

- Instead of spreading out this code in many places (and making your code messy), AOP lets you keep it in one place and apply it wherever needed.

# AOP METAPHORE

Book with sticky notes

# ASPECT ORIENTED PROGRAMMING (AOP)

Steps

- Identify common concerns: find operations that you need to perform in many places within your application, like logging access to methods.
- Write an Aspect: This is like writing your sticky note, where you define what the extra behavior is and when it should happen (before, after, or around the main operation).
- Apply the Aspect: You then tell Spring where and when to apply these sticky notes.
- When your application runs, Spring automatically adds the behavior defined by the aspects to the specified points in your code, without you having to change the code itself.

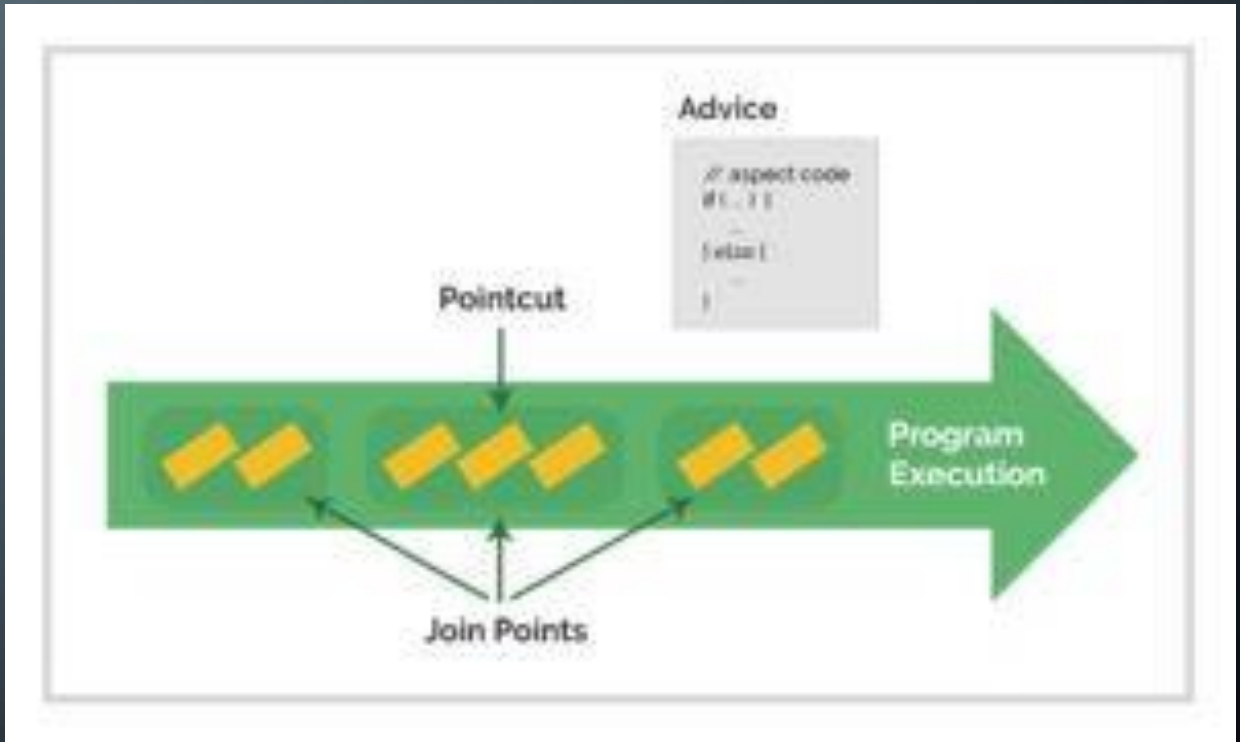
# ASPECT ORIENTED PROGRAMMING (AOP)

So….

- AOP allows you to modularize your concerns and apply them declaratively, making your code cleaner.

# ASPECT ORIENTED PROGRAMMING (AOP)

- General functionalities are injected during runtime when a specified piece of code is called. The general functionalities don't need to be part of the class.




We need to have these definitions clear:

- **Aspect:** A modularization of a concern that cuts across multiple objects. Each aspect focuses on a specific crosscutting functionality
- **Join point:** A point during the execution of a script, such as the execution of a method or property access
- **Advice:** Action taken by an aspect at a particular join point
- **Pointcut:** A regular expression that matches join points. An advice is associated with a pointcut expression and runs at any join point that matches the pointcut

# ANOTHER METAPHOR: THEATRE

- Aspect: director of a play. Controls the production, such as when lights should dim or highlight, when music should start. Aspect is similar: module for cross cutting concerns that can affect multiple points in the application
- Join point: specific moment in the play, change of scenes, climax of a scene, new player entering the stage etc. In software: method call or execution of a block of code. AOP can add behavior to those join points.
- Advice: what the director tells the crew to do at specific moments (turn on the spotlight, close curtains, start music). In software, advice is the action taken by an aspect at that particular join point. Different types of advice: before, after and around
- Pointcut: decision tree or script that the director uses to decided when to trigger specific directions (advices). The criteria that determine whether an advice should be executed or not. In programming, point cuts define the join points of interest and determine when to apply the advice.

# ANOTHER METAPHOR: THEATRE CONT'D

- Target object: main characters in the play who are affected by the decisions of the director.

The objects in our application that are being advised by one or more aspects. They are the focus of the advices applied.

- AOP Proxy: Stagehand who actually dims the light or plays the music based on the instructions of the director. In software, a proxy is an object created to implement the advised behavior on the target object. It wraps the target object and intercepts calls to it, allowing the aspect's advice to be executed at the defined join points.

- Weaving: The rehearsal process where the director's instructions are integrated in the actual performance. In software, the process of applying aspects to target objects to create the advised objects. Can happen at compile time, load time, or during runtime.

# AOP

- What can we think of?

- Logging
- Filtering
- Transactions
- Scheduling
- Caches
- Security
- Etc…

- Sounds familiar?

# SPRING AOP AND ASPECTJ

- There’s multiple implementations of this AOP paradigm.
- The default Spring AOP implementation (which uses our befamed Proxies)
- + Simpler to use than AspectJ, since you don't have to use LTW (load-time weaving) or the AspectJ compiler.
- + It uses the Proxy pattern and the Decorator pattern
- - This is proxy-based AOP, so basically you can only use method-execution joinpoints.
- - Aspects aren't applied when calling another method within the same class.
- - Spring-AOP cannot add an aspect to anything that is not created by the Spring factory
- Or the AspectJ weaving implementation
- + This supports all join points and richer set of pointcuts. This means you can do anything.
- + Because it allows compile-time and load time weaving, typically better performance.
- - Be careful. Check if your aspects are weaved to only what you wanted to be weaved.
- - You need extra build process with AspectJ Compiler or have to setup LTW (load-time weaving)

# SPRING AOP AND ASPECTJ

- Which one should I use??
- https://docs.spring.io/spring/docs/2.5.x/reference/aop.html#aop-choosing
- And you can even combine them ☺!

# SPRING AOP

## Default JDK Dynamic Proxies

- Basically this means all the methods on interfaces can be proxied and reached to attach aspect code to.

- If you really need a rare case to proxy a class in stead of an interface, you can use the CGLIB proxies. This will advice directly to a method that is not declared on an interface.
- https://www.baeldung.com/cglib

# ASPECT J WITH SPRING AOP

Some extra additions to the proxy based support

- The support is intended to be used for objects created outside of the control of any container. Domain objects often fall into this category because they are often created programmatically using the new operator, or by an ORM tool as a result of a database query.

- This is not the 'main' thing for Spring, and is only used in some rare instances… Therefore, the details are out of scope for this Spring training.

# CUSTOM ANNOTATION

Writing our own!

- Let's write an annotation for custom logging what methods are invoked.

- The *@Target* will make our annotation applicable for certain JoinPoints. This is *ElementType.Method,* which means it will only work on methods. If you now use the annotation on anything else then a method, the code will fail to compile. For our case this makes sense, we're logging methods!

- And *@Retention* indicates whether the annotation will be available to the JVM at runtime or not. The default setting is that it is not. This means that Spring AOP will not be able to see the annotation. So we swap it to visible at runtime.

# CUSTOM AOP ASPECT

Writing even more of our own!

- So we have the annotation we can use to indicate that we want to use an aspect on our code. Now let's write the aspect itself.

- This is essentially really nothing more than something with @Aspect and @Component (it needs to be available for the container to detect it as a Spring Bean).

```
package com.example.demo.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class CustomAspect {

    //Aspect logic
}
```

# CUSTOM POINTCUT AND ADVICE

- Now we're doing magic stuff!
- We annotate (@Around, @Before, @After etc) with a pointcut argument. This pointcut states that every annotated LoggerAnnotation should follow this advice.

```
package com.example.demo.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class CustomAspect {

    @Around("@annotation(LoggerAnnotation)")
    public Object logExecutionTime(ProceedingJoinPoint joinPoint) th
        return joinPoint.proceed();
    }
}
```

- (Please beware that there are n different types of pointcuts, if you like these, please use google or the spring documentation on how this works ☺ )
- Make sure that the annotation and this one are in the same package
- The annotated method here is the 'advice'. The input parameter is the ProceedingJoinPoint (ergo; the executing method that has the pointcut annotation we defined above the advice method).
- Now, when the method with the @LoggerAnnotation is called, first this advice will be done.
- Right now it does nothing, just continuing from the pointcut.

# CUSTOM ADVICE

## Implementing the advice

- We can implement 2 advices, one for logging the execution time and one for logging the method calls.

- All we now need is a service with an annotation, and we'll have the results.

```java
package com.example.demo.service;

import com.example.demo.annotation.LoggerAnnotation;

public class CustomService {
    @LoggerAnnotation
    public void serviceMethod() throws InterruptedException {
        Thread.sleep( millis: 3000);
    }
}
```

```java
@Aspect
@Component
public class CustomAspect {

    @Around("@annotation(LoggerAnnotation)")
    public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.currentTimeMillis();

        Object proceed = joinPoint.proceed();

        long executionTime = System.currentTimeMillis() - start;

        System.out.println(joinPoint.getSignature() + " executed in " + executionTime + "ms.");
        return proceed;
    }

    @Before("@annotation(LoggerAnnotation)")
    public Object logMethodName(JoinPoint joinPoint) throws  Throwable {
        System.out.println(joinPoint.getSignature() + " was invoked.");
        return joinPoint;
    }
}
```

# EXERCISE AOP

- Create your own annotation called “LogExecutionTime”
- Before the method starts, log that the method was called and give system time
- After, log that the method was finished (can you print the name of the method? Execution time?)
- Apply this annotation to at least two different methods in your service layer and demonstrate the aspect working by running a test scenario where these methods are invoked.

- Bonus if an exception was thrown, log the class of the exception

# INTRODUCTION TO TRANSACTION MANAGEMENT

# TRANSACTIONS

## What is a Transaction?

- A Transaction is a set of statements that is considered as one. It is only possible for all the statements to take effect. If one of them fails, all the others are rolled back and non of it seems to have happened.

- The transaction can be considered one single action.

- Important when data integrity needs to be secured.

- Transactions should adhere to ACID: Atomicity, Consistency, Isolation and Durability

# TRANSACTIONS

## What does a Transaction look like?

- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform *commit* otherwise *rollback* all the operations.
- What happens in this example? (assume that the changes are committed)
- What happens if you change “maria” with a super lengthy String that exceeds the reserved space for a String in the DB?

```
public void connect() throws SQLException {
    Connection con = DriverManager.getConnection( url: "someurl");
    String insertSQL = "INSERT INTO SOMETABLE (ID, NAME, DATE) VALUES (?,?,?)";
    String updateSQL = "UPDATE SOMETABLE SET NAME = ? WHERE ID = ?";

    PreparedStatement insertStmt = con.prepareStatement(insertSQL);
    PreparedStatement updateStmt = con.prepareStatement(updateSQL);

    insertStmt.setInt( parameterIndex: 1,  x: 10);
    insertStmt.setString( parameterIndex: 2,  x: "maaike");
    insertStmt.setTimestamp( parameterIndex: 3, Timestamp.valueOf("datestring"));
    insertStmt.executeUpdate();

    updateStmt.setString( parameterIndex: 2,  x: "maria");
    updateStmt.setInt( parameterIndex: 1,  x: 22);
    updateStmt.executeUpdate();
}
```

# JDBC TRANSACTIONS

## And with transaction handling

- And what will be the end state of the DB in this example?

```
public void connect() throws SQLException {
    Connection con = DriverManager.getConnection( url: "someurl");
    con.setAutoCommit(false);

    String insertSQL = "INSERT INTO SOMETABLE (ID, NAME, DATE) VALUES (?,?,?)";
    String updateSQL = "UPDATE SOMETABLE SET NAME = ? WHERE ID = ?";

    PreparedStatement insertStmt = con.prepareStatement(insertSQL);
    PreparedStatement updateStmt = con.prepareStatement(updateSQL);

    insertStmt.setInt( parameterIndex: 1,  x: 10);
    insertStmt.setString( parameterIndex: 2,  x: "maaike");
    insertStmt.setTimestamp( parameterIndex: 3, Timestamp.valueOf("datestring"));
    insertStmt.executeUpdate();

    updateStmt.setString( parameterIndex: 2,  x: "maria");
    updateStmt.setInt( parameterIndex: 1,  x: 22);
    updateStmt.executeUpdate();
}
```

# TRANSACTIONS

## Local and Global Transactions

- Local transactions: a transaction that takes place on one transactional resource like a JDBC connection or one message queue. Easy to implement transaction management, can be useful in a centralized system, where all components and resources reside in one place.

- Global transactions: a transaction that uses multiple transactional resources (e.g. multiple relational databases, message queues etc).

- Global transactions are harder for implementing transaction management than local transaction, but they are needed for a distributed system across multiple systems. Transaction management needs to be done across multiple systems so both local and global systems need to be managed.

# SPRING TRANSACTIONS

## Spring transaction abstraction

- Spring Boot detects spring-jdbc and a db (in this case h2) on the classpath.
- It will create a DataSource and a JdbcTemplate for you ready to use.
- Also a DataSourceTransactionManager will be created for you: this is the component that intercepts the @Transactional annotated method.

# @TRANSACTIONAL

## It has some flavours!

- @Transactional can be put on
- An interface (all methods within become transactional);
- a method of an interface;
- a class definition (all public methods within become transactional);
- a public method of a class. (You can safely put this on private methods, but it will simply be ignored)

- @Transactional(readOnly = true)
- This is not necessary, but does some optimizations under the hood for get calls (default is false).

- @Transactional(norollbackFor/rollbackFor Y)
- By default, a transaction will be rolling back on any RuntimeException and Errors but not on checked exceptions (business exceptions). This is how you can rollback on checked exceptions.

# HOW TRANSACTIONS WORK IN SPRING

They use AOP under the hood!

- @Transactional Annotation: declares that the method should be executed within a transactional context.

- AOP: Spring uses proxy-based AOP to wrap the annotated beans with proxy objects that intercept calls to the bean's methods. For classes annotated with @Transactional, Spring creates a proxy that implements the same interface(s) as the class being proxied. When a method annotated with @Transactional is called, the call is intercepted by the proxy.

- Transaction Advice: The proxy uses AOP advice (code that's executed before and/or after method execution). For transaction management, the advice is responsible for starting a new transaction before the method begins (if necessary), and committing or rolling back the transaction after the method executes, based on the execution outcome (success or exception).

# HOW TRANSACTIONS WORK IN SPRING CONT'D

- Transaction manager: The transaction advice communicates with a transaction manager to initiate, commit, or roll back transactions. The transaction manager abstracts the underlying transaction API (like JDBC, JPA, Hibernate, etc.), providing a consistent way to manage transactions across various persistence technologies.

- Isolation and propagation: Through AOP, Spring manages transactional settings such as isolation levels and propagation behavior. Propagation behavior defines how transactions relate to each other, such as whether a method should run within the context of an existing transaction or start a new one. Isolation levels control the degree to which the transaction is isolated from the actions of other transactions.

- Declarative transaction management: This AOP-based approach allows developers to manage transactions declaratively. Instead of having transaction management code spread throughout the application, developers can use annotations (or XML) to declare transactional behavior.

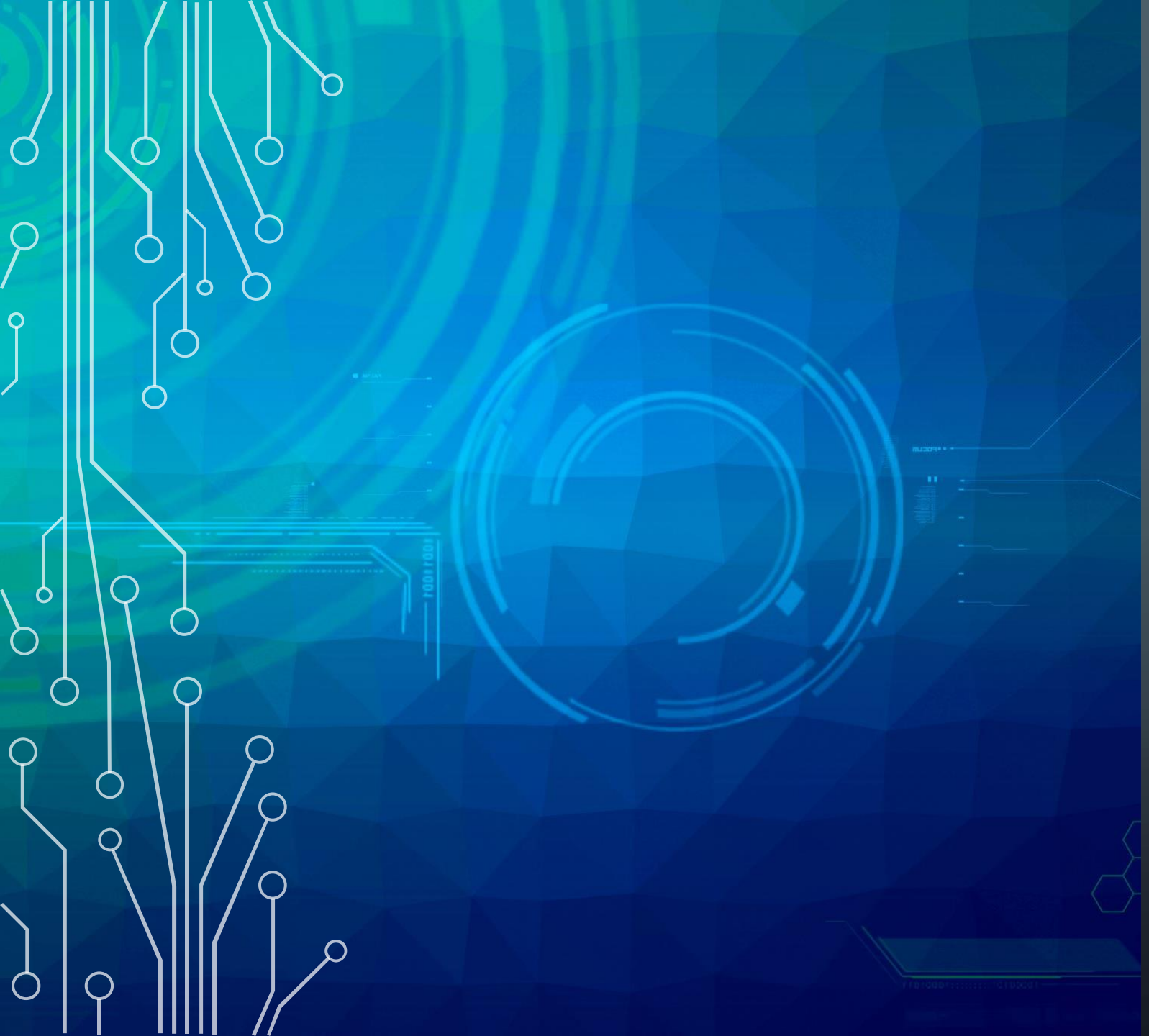
# EXERCISE

- Create two entities: Product and Order.
  - Product has fields for id, name, description, and stockQuantity.
  - Order has fields for id, productId, quantityOrdered, and a status field indicating success or failure
- Define JpaRepository interfaces for both entities to handle CRUD operations.
- Implement a service method placeOrder that accepts a product ID and a quantity to order. This method will perform two operations:
  - Check if the requested quantity is available in stock.
  - If enough stock is available, reduce the stock by the ordered quantity and mark the order as successful.
  - If insufficient stock is available, throw an exception to simulate failure.
- Annotate the placeOrder method with @Transactional to ensure that if any step fails, the stock is not reduced and the order is not placed, maintaining stock integrity.
- Create a controller that uses the service and test the rollback scenarios.
- Bonus: Create a custom annotation @CustomTransactional that allows specifying the transaction isolation level. Use this annotation on the placeOrder method.

# FINAL EXERCISE

Mini-project Smart coffeeshop sytem

- Develop a simulation of a smart coffee shop system that automates the processes involved in order handling, payment processing, and inventory management.
- This project will demonstrate your understanding of Dependency Injection, AOP, and transactional behavior in Spring Boot applications.
- Details in separate file: wrapup-day1.md



# NEXT UP

- ORM
- Security

# QUESTIONS?

- Maaike.vanputten@brightboost.nl
- Or Whatsapp: +31683982426
- Don’t hesitate to contact me, I love to help!