8 Java Annotations interview Questions and Answers

Posted on September 7, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1. Are annotations a compile time or run-time feature?
A1. You can have either compile-time or run-time annotations.

@Override is a simple **compile-time** annotation to catch little mistakes like typing tostring( ) instead of toString( ) in a subclass.

```
public class B extends A {

    private String input;

    @Override
    public String toString(){
        return "input=" + input;
    }
}

```

If you remove the toString( ) method in Class A or misspell toString() method in Class B, the compiler will warn you.

User defined annotations can be processed at compile-time using the **Annotation Processing Tool** (APT) that is included in the Java 6 itself.

@Test is an annotation that JUnit framework uses at **runtime** with the help of reflection to determine which method(s) to execute within a test class.

```
public class MyTest{
    @Test
     public void testEmptyness( ){
         org.junit.Assert.assertTrue(getList( ).
     }

     private List getList( ){
         …
     }
}
```

The test below fails if it takes more than 100ms to execute at runtime.

```
@Test(timeout=100)
public void testTimeout( ) {
    while(true);   //infinite loop
}
```

The code shown below fails if it does not throw "IndexOutOfBoundsException" or if it throws a different exception at runtime. A negative JUnit test.

```
@Test (expected=IndexOutOfBoundsException.class)
public void testOutOfBounds( ) {
       new ArrayList<Object>( ).get(1);
}
```

Q2. Are marker or tag interfaces like Serializable, Runnable, etc obsolete with the advent of annotations (i.e. runtime

16 Technical Key Areas

annotations)?
A2. Everything that can be done with a marker or tag interfaces in earlier versions of Java can now be done with annotations at runtime using reflection. One of the common problems with the marker or tag interfaces like Serializable, Cloneable, etc is that when a class implements them, all of its subclasses inherit them as well whether you want them to or not. You cannot force your subclasses to unimplement an interface. Annotations can have parameters of various kinds, and they're much more flexible than the marker interfaces. This makes tag or marker interfaces obsolete, except for

— In the very rare event of the profiling indicating that the runtime checks are expensive due to being accessed very frequently, and compile-time checks with interfaces is preferred.

— In the event of existing marker interfaces like Serializable, Cloneable, etc are used or Java 5 or later versions cannot be temporarily used.

Q3. What is an annotation, and why are they so popular and used in every framework like Spring, JAX-RS (i.e RESTful web service), JEE 6, and a lot more? When will you favor XML based meta data over annotations based meta data?
A3. One word to explain Annotation is Metadata. Metadata is data about data. So Annotations are metadata for code. The IDEs, compilers, frameworks, and other tools read the annotations to control the behavior of the code that are annotated.

Prior to annotation (and even after) XML were extensively used for metadata, but XML is very verbose and its maintenance was becoming troublesome. Since annotations are closely coupled with the code, they are less verbose. You can define annotations for a class, method, field, etc. XML is defined separately from the code, so it is more verbose as you have to define the class name, method name, etc.

If you want to set some application wide constants and parameters XML would be a better choice because this is not related with any specific piece of code. If you want to expose some method as a Web service, annotation would be a better choice as it needs to be tightly coupled with that method and developer of the method must be aware of this.

Annotations: let you avoid boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to “attribute oriented” (aka declarative) programming. This eliminates the need to maintain “side files” that must be kept up to date with changes in source files.

Annotations based development relieves Java developers from the pain of cumbersome configuration. Annotations provide declarative style programming where the programmer says what should be done and tools emit the code to do it. This assists rapid application development, easier maintenance, and less likely to be bug-prone.

In Spring’s world, an XML config (e.g myapp-applicationContext.xml) to scan for all annotations in the base-package can be defined as shown below.

```
<!-- Component-Scan automatically detects annotat
<context:component-scan base-package="com.myapp.b
```

A POJO is exposed as a RESTful web service with JAX-RS annotations like @Path, @Produces, @GET, @PathParam, etc.

```
package com.mytutorial.webservice;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import com.mytutorial.pojo.User;

@Path("userservice/1.0")
@Produces("application/xml")
public class HelloUserWebServiceImpl implements

  @GET
  @Path("/user/{userName}")
  public User greetUser(@PathParam("userName") S
     User user = new User();
```

```
        user.setName(userName);
        return user;
    }

}

```

In JEE 6 CDI (**C**ontexts and **D**ependency **I**njection) world

You need to have at least an empty "**beans.xml**" file defined under META-INF resource folder for jar files or under WEB-INF folder for war files for DI to take effect.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/n

</beans>

```

and **@inject** annotation to inject dependencies.

```
import javax.inject.Inject;

public class MyApp {

    private final HelloService helloService;

    @Inject
    public MyApp(HelloBean helloBean){
       this.helloService = helloService;
    }

    //.....
}

```

```
@Default
public class HelloServiceImpl implements HelloSer

    public void sayHello() {
        System.out.println("say hello .........")
    }
}

```

Q4. How will you go about defining a custom annotation? Can you give a practical example?

A4. Here is an example where methods annotated with the following custom **@DeadlockRetry** annotation will retry the database operation. The annotation has the attributes *maxTries* and *tryIntervalMillis*.

```
package com.myapp.deadlock;

import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Inherited
public @interface DeadlockRetry
{
    int maxTries() default 10;
    int tryIntervalMillis() default 1000;
}


```

The above custom annotation is annotated with **@Retention** to tell that it is used at run-time, and **@Target** to tell that it is for methods.

Q. Now, who processes this annotation?
A. The following **dynamic proxy** class that uses reflection to see if a method is annotated with @DeadlockRetry. If does, retries the database method call.

```
package com.myapp.deadlock;

import java.lang.annotation.Annotation;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DeadlockRetryHandler implements Inv
{
    private Object target;

    public DeadlockRetryHandler(Object target)
    {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method me
    {
        Annotation[] annotations = method.getAnn
        DeadlockRetry deadlockRetry = (DeadlockR

        final Integer maxTries = deadlockRetry.m
        long tryIntervalMillis = deadlockRetry.t

        int count = 0;

        do
        {
            try
            {
                count++;
                Object result = method.invoke(ta
                return result;
            }
            catch (Throwable e)
            {
                if (!DeadlockUtil.isDeadLock(e))
                {
                    throw new RuntimeException(e
                }

                if (tryIntervalMillis > 0)
                {
                    try
                    {
                        Thread.sleep(tryInterval
                    }
                    catch (InterruptedException
                    {
                        System.out.println("Dead
                    }
                }
            }
        }
        while (count <= maxTries);

        //gets here only when all attempts have
        throw new RuntimeException("DeadlockRetr
                + " due to deadlock in all retry
                new DeadlockDataAccessException(
    }
}
```

The *DeadlockUtil* class determines if the exception is related to deadlock or other SQL exception based on error code and exception type like "LockAcquisitionException".

Now define the target object interface with the custom annotation. The maxTries = 10, tryIntervalMillis = 5000.

```
package com.myapp.engine;

import com.myapp.DeadlockRetry;

public interface AccountServicePersistenceDelega
{
    @DeadlockRetry(maxTries = 10, tryIntervalMil
    abstract Account getAccount(String accountNu
}
```

Q5. What are some of the JAX-RS (i.e RESTful) web service annotations?
A5. **@GET, @POST, @PUT, @DELETE** to specify what type of verb this method (or web service) will perform.

@Produces to specify the type of output this method (or web service) will produce.

@Consumes to specify the MIME media types a REST resource can consume.

@Path to specify the URL path on which this method will be invoked.

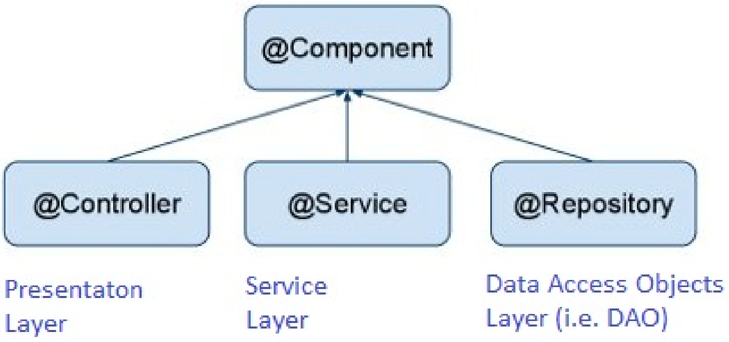
@PathParam to bind REST style parameters to method arguments.

@QueryParam to access parameters in query string (http://localhost:8080/context/accounting-services?accountName=PeterAndCo).

Another example for custom annotation would be for service retry.

Q6. What are some of the widely used Spring annotations?
A6. The Spring beans can be wired either by name or type. **@Autowired** by default is a type driven injection. @Autowired is Spring annotation, while @Inject is a JEE CDI annotation. **@Inject** is equivalent to @Autowired or @Autowired(required=true). **@Qualifier** spring annotation can be used to further fine-tune auto-wiring. There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

The different POJO objects in different layers are annotated with one of the following key annotations.



Spring annotations

@Component is the parent annotation from which the other annotations like **@Service**, **@Resource**, **@Repository** etc are defined. Here is an example of a DAO class annotated with @Repository, and @Resource is used to inject “jdbcTemplateSybase” with which the database calls are made.

```
@Repository(value = "myapp_Dao")
public class CashForecastDaoImpl implements Cash
{

    @Resource(name = "myapp_JdbcTemplate")
    private JdbcTemplate jdbcTemplateSybase;//co

    public PortfolioSummaryVO  retrievePortfolio
      //............use jdbcTemplateSybase
    }

}
```

The annotations shown above allow you to declare beans that are to be picked up by autoscanning with or **@ComponentScan**.

The **@Configuration** annotation was designed as the replacement for XML configuration files. You use @Bean annotation to wire up dependencies. Here is an example.

```
package com.myapp.bdd.stories;

import com.myapp.*;

import org.powermock.api.mockito.PowerMockito;
import org.springframework.context.annotation.Be
import org.springframework.context.annotation.Co
import org.springframework.context.annotation.Co
import org.springframework.context.annotation.Fi
import org.springframework.context.annotation.Im

@Configuration
//packages and components to scan and include
@ComponentScan(
        basePackages =
        {
            "com.myapp.calculation.bdd",
            "com.myapp.refdata",
            "com.myapp.dm.dao"

        },
        useDefaultFilters = false,
         //interfaces
        includeFilters =
        {
            @ComponentScan.Filter(type = FilterT
            @ComponentScan.Filter(type = FilterT
            @ComponentScan.Filter(type = FilterT
            @ComponentScan.Filter(type = FilterT

            @ComponentScan.Filter(type = FilterT
        })
//import other config classes
@Import(
{
    StepsConfig.class,
    DataSourceConfig.class,
    JmsConfig.class,
    PropertiesConfig.class
})
public class StoryConfig
{
    //creates a partially mocked transaction ser
    @Bean(name = "txnService")
    public TransactionService getTransactionServ
    {
        return PowerMockito.spy(new TransactionS
    }
}

```

Q7. What are some of the widely used JEE CDI annotations?
A7. CDI is one of the biggest promises of JEE 6.

@Default, **@Alternative**, and **@Name**: A class is @Default by default. Thus marking it a redundant annotation. Mark as @Alternative to annotate the alternative implementations that implement the same interface. Use the @Named annotation to look up by name. The **@Named** annotation is also used by JEE 6 application to make the bean accessible via the Unified EL.

The interface

```
public interface PaymentService {
    public void processPayment(BigDecimal amount,
}
```

The **default** implementation

```
@Default
public class CashPaymentServiceImpl implements Pa
    public void processPayment(BigDecimal amount,
        //.............
    }
}

```

An **alternative** implementation

```
@Alternative
public class BPayPaymentServiceImpl implements Pa
    public void processPayment(BigDecimal amount,
        //.............
    }
}
```

```
@Alternative
public class CreditCardPaymentServiceImpl impleme
    public void processPayment(BigDecimal amount,
        //.............
    }
}
```

An alternative can be selected via XML based deployment files in {classpath}/META-INF/beans.xml

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/n
    <alternatives>
        <class>BPayPaymentServiceImpl</class>
    </alternatives>
</beans>

```

A **named** implementation

```
@Named("cash")
public class CashPaymentServiceImpl implements Pa
    public void processPayment(BigDecimal amount,
        //.............
    }
```

```
}
```

Named annotations can be looked up via the JEE bean container.

```
public class PaymentMain {
    //...

    public static void main(String[] args) throw
        PaymentService svc = (PaymentService) be
                .getBeanByName("cash");

        svc.processPayment(new BigDecimal("9.00"
    }

}
```

@Inject to inject via fields and constructors.

```
public class PaymentMain {

    @Inject
    private PaymentService paymentService;
    //...
}
```

```
public class PaymentMain {

    private PaymentService paymentService;

    @Inject
    public PaymentMain(PaymentService paymentSer
         this. paymentService = paymentService;
    }

    //....
}
```

@Produces for more complex object construction via **factory methods**.

```
import javax.enterprise.inject.Produces;

public class PaymentFactory {

    @Produces
    public PaymentService createPaymentService()
        return new CashPaymentServiceImpl();
    }
    //…...
}

```

@Qualifier is required when an interface has multiple implementations to choose the one you want to inject. All objects and producers in CDI have qualifiers. If you do not assign a qualifier, by default has the qualifier **@Default** and **@Any**. So, if you don't specify a qualifier, you will be assigned one.

Meta meta annotations to define a qualifier

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*
import static java.lang.annotation.RetentionPoli

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface BPay {

}
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*
import static java.lang.annotation.RetentionPoli

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface CreditCard {

}
```

Use the qualifier annotations

```
@BPay
public class BPayPaymentServiceImpl implements Pa
    public void processPayment(BigDecimal amount,
        //.............
    }
}
```

```
@CreditCard
public class CreditCardPaymentServiceImpl impleme
    public void processPayment(BigDecimal amount,
        //.............
    }
```

```
}
```

BPay is injected

```
public class PaymentMain {

    private PaymentService paymentService;

    @Inject
    public PaymentMain(@BPay PaymentService paym
         this. paymentService = paymentService;
    }

    //....
}
```

Multiple types can be injected into the same bean

```
public class PaymentMain {
    //...

    @Inject @BPay
    private PaymentService bpayService;

    @Inject @CreditCard
    private PaymentService ccService;

    private PaymentService paymentService;

    //....

    @PostConstruct
    protected void init() {
      if(account.isBPay()) {
         this.paymentService = this.bpayService;
      }
      else {
         this.paymentService = this.ccService;

      }
    }
}
```

Note: @PostConstruct annotation is used to decide which service to use. Alternatively, you can use a factory method with **@Produces** annotation.

```
@Produces
public PaymentService createPayment (@BPay Payme
                        @CreditCard PaymentServi

   if (account.isBPay()) {
      return bpayService;
   } else {
       return ccService;
   }
}
```

Q8. How would you unit test CDI with JUnit?
A8. **@RunWith** annotation and pass “CdiRunner.class”.

```
@RunWith(CdiRunner.class)
class PaymentServiceTest {
   @Inject
   PaymentService paymentService;
   //.....
}
```

There are other annotations like @AdditionalClasses, @AdditionalPackages, @AdditionalClasspath, @Produces, @EnabledAlternatives, @ProducesAlternative, etc and scoping annotations like @InRequestScope, @InSessionScope, and @InConversationScope.

Q9. In Servlet 3.0, why is configuring your servlet via deployment descriptor file web.xml optional?
A9. Servlets 3.0 have come up with a set of new Annotations for the declarations of Servlet Mappings, Init-Params, Listeners, and Filters to make the code more readable by making the use of Deployment Descriptor (web.xml) absolutely optional.

```
@WebServlet(
  asyncSupported = false,
  name = "AccountingServlet",
  urlPatterns = { "/acount" },
  initParams = {
    @WebInitParam(name = "param1", value = "valu
    @WebInitParam(name = "param2", value = "valu
  }
)
public class AccountingServlet extends HttpServl
     //...
}

```

Servlet filters, listeners, etc can be configured with annotations.

Popular Posts

♦ 11 Spring boot interview questions & answers

823 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

765 views

18 Java scenarios based interview Questions and Answers

399 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

388 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

295 views

♦ 7 Java debugging interview questions & answers

293 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

285 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

279 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

239 views

001B: ♦ Java architecture & design concepts interview questions & answers

201 views

Bio **Latest Posts**

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ ♦ 16 FAQ JMS interview questions & answers

02: ♥♦ 8 real life examples of SAR technique for Java developers ›

Posted in Annotations, member-paid

Tags: Core Java FAQs, Java/JEE FAQs

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from "Blah blah" to "Wow"? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼