02

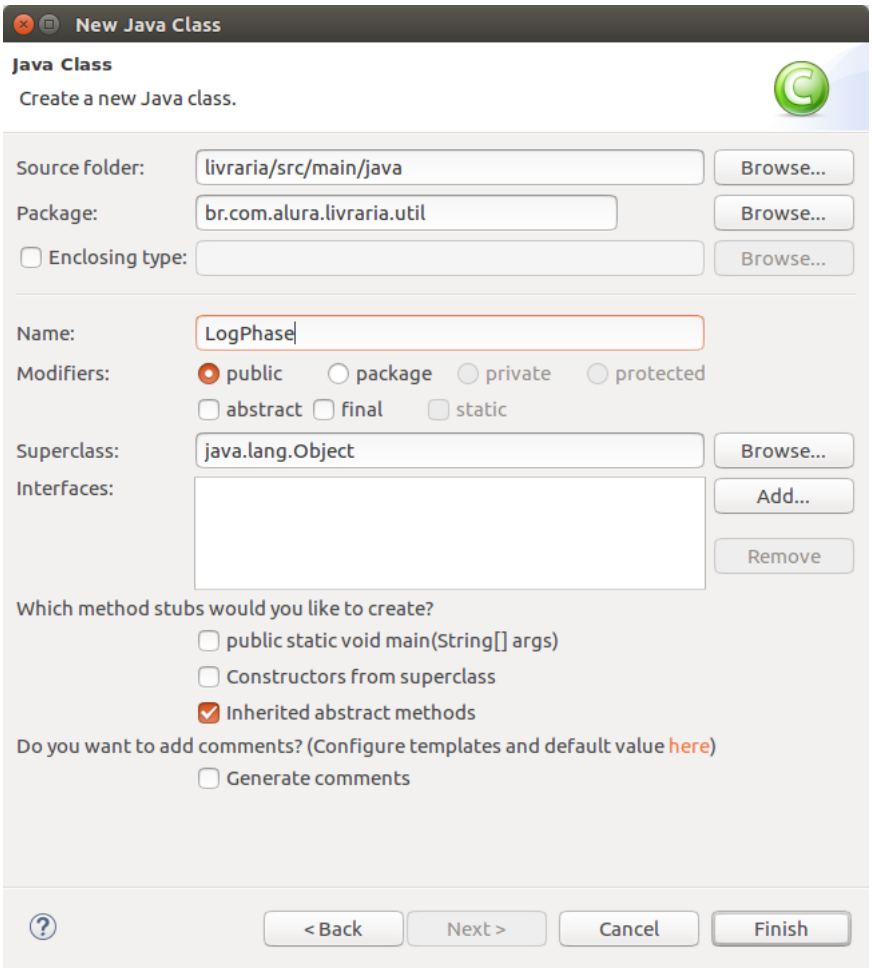
Mais sobre observers

Transcrição

Agora falta implementar a parte de quem vai ficar esperando receber as notificações. Para isso vamos implementar o *design pattern observer*.

Alguém irá ficar observando um determinado evento e quando o evento for disparado, a parte do sistema que tem interesse executar o que for necessário. Para isto, será necessário criar uma classe.

Para isso no projeto `livraria`, foi criada uma classe chamada `LogPhase`, no pacote `br.com.alura.livraria.util`:



Nesta classe, foi criado um método `void`, que recebe como parâmetro o tipo que desejamos observar. Nesse caso um `PhaseEvent`. O método apenas imprime uma mensagem no Console, com o id da fase:

```
public class LogPhase {

    public void log(@Observes PhaseEvent phaseEvent) {
        System.out.println("FASE: " + phaseEvent.getPhaseId());
    }
}
```

No `PhaseListenerGenerico`, é necessário notificar quem está observando o evento. Para fazer isso, é utilizado um objeto do CDI chamado `Event` - atrelado ao objeto que está sendo observado.

Vamos receber o `Event` via injeção de dependências, e nos métodos vamos disparar o evento, por meio da chamada ao método `fire()`, que recebe com argumento o `PhaseEvent`.

```
public class PhaseListenerGenerico implements PhaseListener {

    private static final long serialVersionUID = 4332180564534271099L;

    @Inject
    private Event<PhaseEvent> observer;

    @Override
    public void afterPhase(PhaseEvent event) {
        observer.fire(event);
    }

    @Override
    public void beforePhase(PhaseEvent event) {
        observer.fire(event);
    }

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}
```

Da forma que implementamos, o primeiro ponto é que o evento será disparado antes e depois da *phase*, então o `LogPhase` irá imprimir a mensagem duas vezes no console. Queremos que ele imprima a mensagem apenas uma vez.

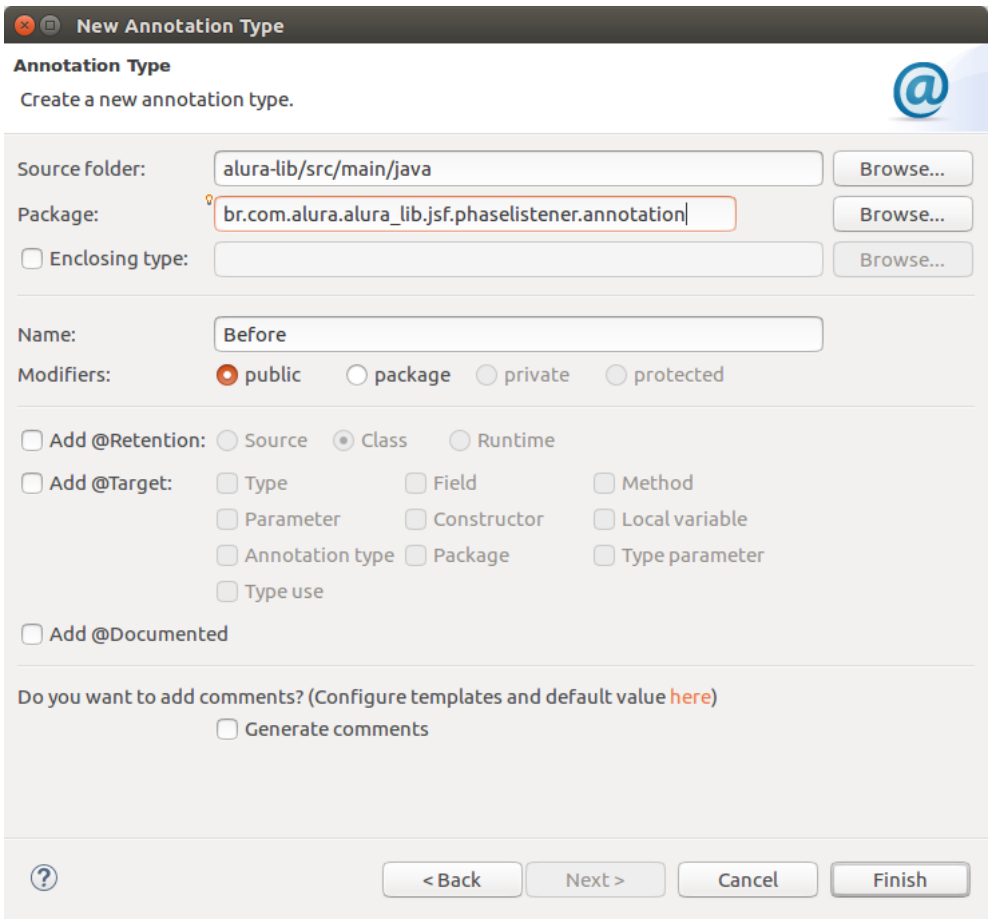
Mais ainda: queremos poder observar o objeto em um determinado momento, que pode ser antes ou depois. E para uma determinada *phase*.

Se observarmos o `Autorizador` (pacote `br.com.alura.livraria.util`), queremos observar apenas o `afterPhase` para a *phase* `br.com.alura.livraria.util`. No `LogPhase`, poderíamos fazer algo como:

```
public void log(@Observes @After @RestoreView PhaseEvent phaseEvent) {
    System.out.println("FASE: " + phaseEvent.getPhaseId());
}
```

Nos *observer*, também é possível utilizar anotações para qualificar o evento observado. Será necessário criar as anotações `@After` e `Before`, bem como uma anotação para cada fase.

Vamos iniciar criando a anotação `@Before`, no pacote `br.com.alura.alura_lib.jsf.phaselistener.annotation`:



A anotação `@Before` é um qualificador:

```
@Qualifier
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Before {
}
```

Dentro do mesmo pacote, foi criada a anotação `@After`, para indicar que se deseja observar o `afterPhase()`:

```
@Qualifier
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface After {
}
```

No `PhaseListenerGenerico`, no momento em que estamos disparando o evento, é necessário indicar qual momento estamos notificando. Isso é possível por meio do método `select()`, onde passamos um qualificador. Não podemos passar a anotação diretamente como argumento para um método. Por esse motivo, é necessário utilizar a classe abstrata `AnnotationLiteral`. Por ser abstrata, precisamos implementar a classe, mas não é necessário escrever algum código dentro dela, no nosso caso:

```
@Override
public void afterPhase(PhaseEvent event) {
    observer
        .select(new AnnotationLiteral<After>() {})
        .fire(event);
```

```
    }

    @Override
    public void beforePhase(PhaseEvent event) {
        observer
            .select(new AnnotationLiteral<Before>() {})
            .fire(event);
    }
```

Perceba que dentro dos métodos, o código é bem similar, o que muda é apenas a anotação que estamos passando. Esse comportamento será isolado em uma classe chamada `PhaseListenerObserver`, no pacote `br.com.alura.alura_lib.jsf.phaselistener`.

Vamos criar uma interface fluente, onde os métodos `after()` e `before()` retornam um próprio objeto. O atributo `momento` é a anotação que representa o momento a ser observado. E o método `fire()` dispara o evento para o momento.

```
public class PhaseListenerObserver {

    @Inject
    private Event<PhaseEvent> observer;
    private Annotation momento;

    public PhaseListenerObserver after() {
        this.momento = new AnnotationLiteral<After>(){};
        return this;
    }

    public PhaseListenerObserver before() {
        this.momento = new AnnotationLiteral<Before>(){};
        return this;
    }

    public void fire(PhaseEvent phaseEvent) {
        observer
            .select(momento)
            .fire(phaseEvent);
    }
}
```

Na classe `PhaseListenerGenerico`, vamos receber o `PhaseListenerObserver`, em vez de receber um `Event`. Nos métodos `afterPhase()` e `beforePhase()`, basta realizar a chamada aos método `after()` e `before()`, disparando o evento em seguida, por meio do método `fire()`.

```
public class PhaseListenerGenerico implements PhaseListener {

    private static final long serialVersionUID = 4332180564534271099L;

    @Inject
    private PhaseListenerObserver observer;

    @Override
    public void afterPhase(PhaseEvent event) {
        observer
```

```
                .after()
                .fire(event);
        }

        @Override
        public void beforePhase(PhaseEvent event) {
            observer
                .before()
                .fire(event);
        }

        // getPhaseId()
    }
```

Agora resta criar os qualificadores para as fases. Mas no JSF, temos seis fases, então seria uma anotação para cada fase. Nesse caso vamos criar uma anotação que contém um atributo representando qual fase será observada.

No pacote `br.com.alura.alura_lib.jsf.phaselistener.annotation` , criamos uma anotação chamada `Phase` . Queremos ter um `value()` do tipo `PhaseId` . Mas o código a seguir não compila:

```
@Qualifier
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Phase {

    PhaseId value();
}
```

Isso ocorre porque o atributo da anotação tem que ser um tipo primitivo, uma `String` , uma `Enum` , uma anotação ou um `array` de uma dimensão. Para resolver o problema, vamos criar um `Enum` dentro da anotação, onde serão listadas todas as fases do JSF.

Vamos passar cada uma dessas fases para uma atributo da `Enum` :

```
@Qualifier
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Phase {

    Phases value();

    enum Phases {
        RESTORE_VIEW(PhaseId.RESTORE_VIEW),
        APPLY_REQUEST_VALUES(PhaseId.APPLY_REQUEST_VALUES),
        PROCESS_VALIDATIONS(PhaseId.PROCESS_VALIDATIONS),
        UPDATE_MODEL_VALUES(PhaseId.UPDATE_MODEL_VALUES),
        INVOKE_APPLICATION(PhaseId.INVOKE_APPLICATION),
        RENDER_RESPONSE(PhaseId.RENDER_RESPONSE);

        private PhaseId phaseId;

        Phases(PhaseId phaseId) {
            this.phaseId = phaseId;
```

```
        }

        public PhaseId getPhaseId() {
            return phaseId;
        }
    }
}
```

No projeto `livraria`, na classe `LogPhase`, vamos passar a utilizar o seguinte código:

```
public void log(@Observes @After @Phase(Phases.RESTORE_VIEW) PhaseEvent phaseEvent) {
    System.out.println("FASE: " + phaseEvent.getPhaseId());
}
```

No `PhaseListenerObserver`, na biblioteca, vamos adicionar um novo `select()`, para utilizar a anotação `@Phase`:

```
public void fire(PhaseEvent phaseEvent) {
    observer
        .select(momento)
        .select(new AnnotationLiteral<Phase>(){})
        .fire(phaseEvent);
}
```

Mas como foi instanciado um `AnnotationLiteral`, não temos o valor do atributo `value`. Vimos que `AnnotationLiteral` é uma classe abstrata, então para utilizarmos de uma forma mais natural, instanciando um objeto, vamos criar uma nova classe. Essa classe se chamará `PhaseLiteral` e ficará no pacote `br.com.alura.alura_lib.jsf.phaselistener.annotation`.

No `PhaseListenerObserver`, queremos instanciar o `PhaseLiteral` e passar a fase que desejamos observar. Obtemos a fase por meio do `PhaseEvent`:

```
public void fire(PhaseEvent phaseEvent) {
    observer
        .select(momento)
        .select(new PhaseLiteral(phaseEvent.getPhaseId()))
        .fire(phaseEvent);
}
```

A classe `PhaseLiteral` precisa extender de `AnnotationLiteral` e implementar a anotação `@Phase` (que é uma interface [`public @interface Phase`]). Dessa forma vamos ter que implementar o método `value()`.

Vamos adicionar o construtor que recebe o `phaseId`:

```
public class PhaseLiteral extends AnnotationLiteral<Phase> implements Phase {

    private static final long serialVersionUID = 3320747441506123437L;

    public PhaseLiteral(PhaseId phaseId) {

    }
```

```
    @Override
    public Phases value() {
        return null;
    }
}
```

Agora precisamos retornar uma fase. A `enum Phases` possui seus valores iguais aos nomes da fases do JSF, podemos obter a `Phase` por meio do método `valueOf()`, passando o `phaseId.getName()`:

```
public class PhaseLiteral extends AnnotationLiteral<Phase> implements Phase {

    private static final long serialVersionUID = 3320747441506123437L;

    private Phases phases;

    public PhaseLiteral(PhaseId phaseId) {
        phases = Phase.Phases.valueOf(phaseId.getName());
    }

    @Override
    public Phases value() {
        return phases;
    }
}
```

Agora precisamos instalar o `alura-lib` no repositório local para que possamos testar as alterações. Em seguida rodamos um "Maven > Update Project..." no projeto `livraria`.

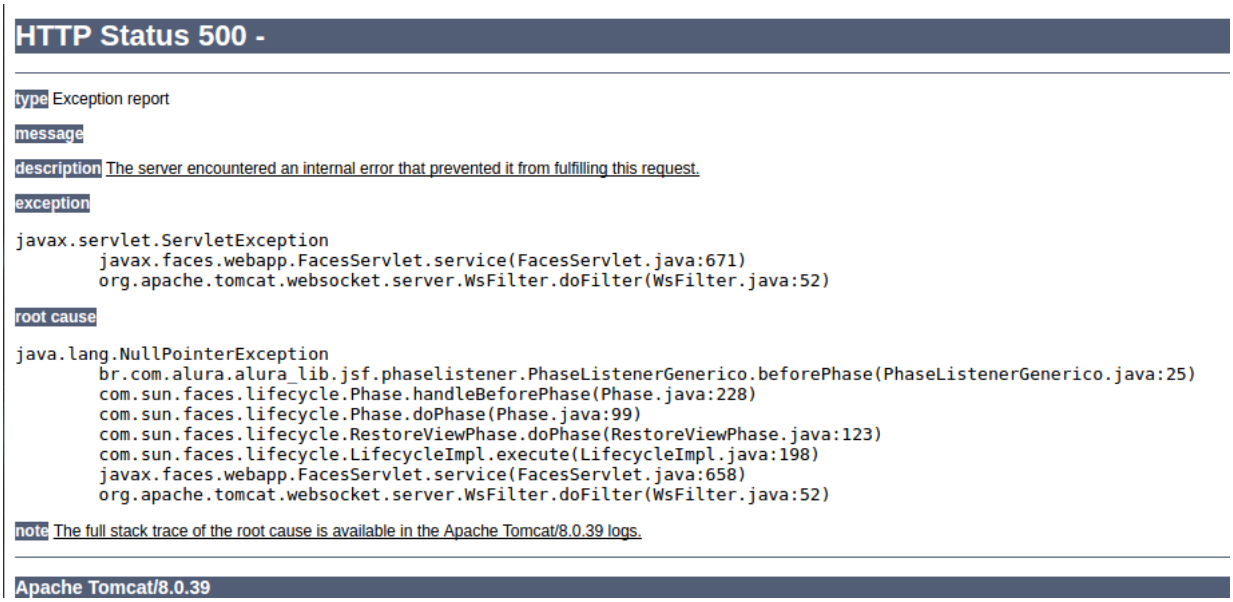
A classe `LogPhase` ainda não compilou, pois faltam os *imports*. Para isso basta utilizar o atalho "Ctrl + Shift + O".

O que acontece se removermos o qualificador `@Phase`, por exemplo? Simplesmente será observador o *after* e todos os eventos, já que não estamos sendo específicos. Vamos remover e ver o resultado:

```
public class LogPhase {

    public void log(@Observes @After PhaseEvent phaseEvent) {
        System.out.println("FASE: " + phaseEvent.getPhaseId());
    }
}
```

Por fim, fazemos um "Clean" e iniciamos o Tomcat. Mas recebemos um erro. Um `NullPointerException` na classe `PhaseListenerGenerico`, indicando que o atributo `observer` está nulo:



HTTP Status 500 -

type Exception report

message

description The server encountered an internal error that prevented it from fulfilling this request.

exception

```
javax.servlet.ServletException
        javax.faces.webapp.FacesServlet.service(FacesServlet.java:671)
        org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
```

root cause

```
java.lang.NullPointerException
        br.com.alura.alura_lib.jsf.phaselistener.PhaseListenerGenerico.beforePhase(PhaseListenerGenerico.java:25)
        com.sun.faces.lifecycle.Phase.handleBeforePhase(Phase.java:228)
        com.sun.faces.lifecycle.Phase.doPhase(Phase.java:99)
        com.sun.faces.lifecycle.RestoreViewPhase.doPhase(RestoreViewPhase.java:123)
        com.sun.faces.lifecycle.LifecycleImpl.execute(LifecycleImpl.java:198)
        javax.faces.webapp.FacesServlet.service(FacesServlet.java:658)
        org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
```

note The full stack trace of the root cause is available in the Apache Tomcat/8.0.39 logs.

Apache Tomcat/8.0.39

Mas estamos injetando o atributo! O que pode estar acontecendo? Acontece que a classe `PhaseListenerGenerico` é gerenciada pelo JSF, e não pelo CDI. Na próximo aula vamos ver como resolver esse problema.