Transcrição

Bem vindo de volta! Nesse vídeo iremos melhorar ainda mais o nosso Servlet. Esse conteúdo, inclusive, vai um pouco além do escopo do curso, que é focar os recursos do Servlet. Porém, temos um momento oportuno para estudarmos boas práticas de código.

Vamos dar uma olhada no nosso controlador:

```
@WebServlet("/entrada")
public class UnicaEntradaServlet extends HttpServlet {
    private static final long serialVersio
![ibknkl](https://s3.amazonaws.com/caelum-online-public/1001-servlet-autenticac
nUID = 1L;


    protected void service(HttpServletRequest request, HttpServletResponse resp

        String paramAcao = request.getParameter("acao");

        String nome = null;

        if(paramAcao.equals("ListaEmpresas")) {

            ListaEmpresas acao = new ListaEmpresas();
            nome = acao.executa(request, response);

        } else if(paramAcao.equals("RemoveEmpresa")) {

            RemoveEmpresa acao = new RemoveEmpresa();
            nome = acao.executa(request, response);

        } else if(paramAcao.equals("MostraEmpresa")) {

            MostraEmpresa acao = new MostraEmpresa();
            nome = acao.executa(request, response);

        } else if (paramAcao.equals("AlteraEmpresa")) {

            AlteraEmpresa acao = new AlteraEmpresa();
            nome = acao.executa(request, response);

        } else if (paramAcao.equals("NovaEmpresa")) {

            NovaEmpresa acao = new NovaEmpresa();
            nome = acao.executa(request, response);
        } else if (paramAcao.equals("NovaEmpresaForm")) {

            NovaEmpresaForm acao = new NovaEmpresaForm();
            nome = acao.executa(request, response);
        }

        String[] tipoEEndereco = nome.split(":");
        if(tipoEEndereco[0].equals("forward")) {
            RequestDispatcher rd = request.getRequestDispatcher("WEB-INF/view/"
            rd.forward(request, response);
        } else {
            response.sendRedirect(tipoEEndereco[1]);

        }
    }

}
```

O problema desse código é que temos várias condições `if` e `else`, o que remete à programação procedural. O primeiro defeito é que sempre precisaremos dar manutenção a essas condições, já que temos um código que nunca para de crescer (ou diminuir) quando criamos ou alteramos alguma ação.

Nosso objetivo é termos um código genérico e mais orientado a objeto. Repare que nossas condições sempre seguem o mesmo padrão: instancia a ação, chama o método `executa()`. Será que é possível generalizar esse código para instanciar a ação e chamar o método `executa()`? Mas é claro!

Repare também que nossas strings têm o mesmo nome das classes que criamos. Isso facilita o nosso trabalho (e foi feito de propósito), pois se recebemos a string `ListaEmpresas`, a ação a ser executada deve ser `ListaEmpresas`.

O que gostaríamos de fazer é criar um objeto da classe que representa a ação e chamar o método `executa()`, passando `request` e `response`. O código a seguir não funciona, mas serve para ilustrar nosso objetivo:

```
paramAcao.executa(request, response)
```

Vamos trabalhar?

Se queremos chamar o método `executa()`, primeiro precisamos do objeto. Se queremos o objeto, primeiro precisamos ter a classe. Portanto, nossa prioridade agora é saber com qual classe estamos lidando. Para isso, criaremos uma `String` chamada `nomeDaClasse`.

O nome da classe é o mesmo do `paramAcao`, certo? Na verdade, se repararmos bem nos imports, o nome da classe não é somente `ListaEmpresas` ou `RemoveEmpresa`, por exemplo, mas sim o nome do pacote acompanhado da respectiva da classe - ou seja, teremos que concatenar `br.com.alura.gerenciador.acao.` juntamente com `paramAcao`:

```
String nomeDaClasse = "br.com.alura.gerenciador.acao." + paramAcao;
```

Desse modo, temos uma `string` maior que representa o nome qualificado da classe. Agora precisamos instruir a máquina virtual acarregar a própria classe a partir do nome da classe. Como fazemos isso?

Existe uma classe chamada `Class` - estranho, não? Como ela é uma classe do pacote java.lang, não precisaremos importar nada. Essa classe possui o método `forName()`, que recebe uma string. Essa string deve ser o `nomeDaClasse`:

```
Class.forName(nomeDaClasse);//carrega a classe com o nome
```

Apesar da linha estar correta, nosso código ainda não irá compilar, pois está repleto de exceções (que iremos ignorar por enquanto). Nesse ponto, quando usamos uma classe, a nossa maquina virtual carrega a classe automaticamente, apenas uma vez, e deixa essa classe na memória.

Agora queremos que a máquina virtual devolva a classe:

```
Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nome
```

Ou seja, se o nosso `paramAcao` é `ListaEmpresas`, ela irá carregar `ListaEmpresas` e irá retornar alguma representação dessa classe na memória. Isso não é um objeto, mas sim o código compilado na memória que a máquina virtual carregou. Agora vamos instanciar o objeto com base nessa classe.

Repare que temos disponível uma API completa a partir dessa classe, como `getInterface()`. `getPackage()` e `getConstructors()`. Essa API se chama **Reflection**, e ela é tão importante que até o momento temos dois cursos sobre ela na Alura

Nesse caso, queremos criar uma instância a partir dessa classe. Faremos isso com o método `newIstance()`:

```
Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nome
classe.newInstance();
```

Quando usamos esse método, ele aparece riscado. Isso significa que o método está deprecado (fora de uso) e talvez seja removido do Java em uma versão futura. Porém, ele continua funcionando e sendo utilizado em milhares de projetos.

O correto seria pedirmos o construtor da classe e pedirmos uma nova instância a partir dele, que é o que fazemos nas estruturas condicionais (`if()` e `else if()`) do nosso código. Porém, aqui, vamos usar o atalho `newInstance()`.

O método `newInstance` cria um objeto:

```
Object obj = classe.newInstance();
```

Porém, recebemos uma referência genérica, pois o `newInstance` não sabe se a nossa classe é uma `ListaEmpresa`, uma string, um Servlet ou outra coisa, por isso ele retorna um `obj`.

Apesar de estar correto, nosso código continua não compilando por causa das exceções.

Agora que criamos o objeto, devemos chamar o método `executa()`. Vamos fazer isso:

```
Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nome
Object obj = classe.newInstance();
String nome = obj.executa(request, response);
```

Dessa vez nosso código não irá compilar por outro motivo: nós temos uma referência do tipo `Object`, mas quem define quais métodos podemos chamar é a referência. Como nossa classe `Object` não possui o método `executa()` que recebe `request,response`, o código não compila.

Repare que todas as nossas ações possuem o mesmo método `executa` que recebe `request,response`. Ou seja, elas têm um comportamento que já é esperado e que compartilham entre si. Vamos explicitar esse comportamento através da implementação de uma interface, que chamaremos de `Acao`. Faremos isso na nossa classe `ListaEmpresas`:

```
public class ListaEmpresas implements Acao{

    public String executa(HttpServletRequest request, HttpServletResponse respo

        System.out.println("listando empresas");

        Banco banco = new Banco();
        List<Empresa> lista = banco.getEmpresas();

        request.setAttribute("empresas", lista);

        return "forward:listaEmpresas.jsp";
    }

}
```

Agora criaremos essa interface clicando no ícone de erro que aparece na linha `public class ListaEmpresas implements Acao{` e em seguida em `Create interface 'Acao'`. Como queremos que essa interface seja criada no pacote `acao`, basta clicarmos em "Finish".

Nessa nova interface, vamos colar a assinatura do método anterior:

```
public interface Acao {

    public String executa(HttpServletRequest request, HttpServletResponse respo

}
```

Como temos um método abstrato, ele é automaticamente público e não precisaremos colocar isso no código. Vamos repetir o processo de `ListaEmpresas` em todas as nossas ações, por exemplo:

```
public class AlteraEmpresa implements Acao {

    public String executa(HttpServletRequest request, HttpServletResponse respo

        System.out.println("Alterando empresa");

        String nomeEmpresa = request.getParameter("nome");
        String paramDataEmpresa = request.getParameter("data");
        String paramId = request.getParameter("id");
        Integer id = Integer.valueOf(paramId);
//...
}
```

Nesse ponto, todas as nossas ações implementam formalmente uma interface que se chama `Acao` e define o método `executa()`.

Agora precisaremos fazer um cast para transformarmos nossa referência genérica (`obj`) em uma referência específica (`acao`), e utilizarmos essa referência para chamar o método `executa()`:

```
    Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nome
    Object obj = classe.newInstance();
    Acao acao = (Acao) obj;
    String nome = acao.executa(request, response);
```

Dessa forma, podemos comentar todos os `if` e `else if` que criamos para nossas classes. Para simplificar, vamos reorganizar o código que estamos utilizando. Além disso, vamos refazer nosso cast em apenas uma linha. Assim, teremos:

```
@WebServlet("/entrada")
public class UnicaEntradaServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;


    protected void service(HttpServletRequest request, HttpServletResponse resp

        String paramAcao = request.getParameter("acao");

        String nomeDaClasse = "br.com.alura.gerenciador.acao." + paramAcao;

        Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nom
        Acao acao = (Acao) classe.newInstance();
        String nome = acao.executa(request, response);



        String[] tipoEEndereco = nome.split(":");
        if(tipoEEndereco[0].equals("forward")) {
            RequestDispatcher rd = request.getRequestDispatcher("WEB-INF/view/"
            rd.forward(request, response);
        } else {
            response.sendRedirect(tipoEEndereco[1]);

        }
    }
}
```

Para resolvermos as exceções, selecionaremos essa parte do código...

```
Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nome
Acao acao = (Acao) classe.newInstance();
String nome = acao.executa(request, response);
```

...e clicaremos com o botão direito, depois em "Sorround With > Try/multi-catch Block". Precisamos fazer o `Try/catch` pois o nosso método só pode jogar `ServletException` e `IOException`, e, por causa da herança, não conseguimos adicionar novas exceções.

Como não estamos interessados em tratamento de exceções nesse momento, vamos criar um `throw new ServletException(e)` (que leva à exceção original). Dessa forma, teremos:

```
String nome;
try {
    Class classe = Class.forName(nomeDaClasse);//carrega a classe com o nome
    Acao acao = (Acao) classe.newInstance();
    nome = acao.executa(request, response);
} catch (ClassNotFoundException | InstantiationException | IllegalAccessExcepti
    throw new ServletException(e);
}
```

Agora já temos um belo código - ainda que menos relacionado ao mundo Servlet, e mais relacionado a padrões de projeto e, principalmente, Reflection. Para quem ficou com duvidas, existem cursos de Reflection aqui na Alura que ensinam em detalhes esse conceito.

Repare que nossas ações definem apenas um método. No mundo de padrões de projeto, essas classes que encapsulam a execução de algo são chamadas de **comandos**. Normalmente os comandos têm métodos como `execute()`, `run()`, `call()`, entre outros.

Para testarmos se o nosso código continua funcionando, basta rodarmos o Tomcat novamente e acessarmos as respectivas URLs de nossas ações. Aparentemente nosso Servlet continua funcionando, e, se criarmos uma nova ação, não precisaremos mais alterar o controlador `UnicaEntradaServlet`, desde que sigamos algumas convenções:

- precisaremos criar a nova ação dentro do nosso pacote `acao`, do contrário ela não irá funcionar
- precisaremos implemenetar a interface `Acao`, do contrário o cast que criamos também não irá funcionar

Aperfeiçoamos bastante o nosso controlador, que agora é bem genérico e profissional. Ainda é possível melhorá-lo e existem controladores já prontos que são mais sofisticados, mas objetivo aqui era conseguirmos escrever um controlador genérico e termos uma noção de como usá-lo para aplicarmos essas noções em controladores como Spring MVC ou VRaptor.

No próximo vídeo, faremos um resumo e analisaremos o padrão que implementamos nesse projeto. Não se esqueça de resolver os exercícios e até a próxima!