02

Mais flexibilidade ao gerar conteúdo

Download

Você pode fazer o download do projeto deste capítulo [aqui (https://s3.amazonaws.com/caelum-online-public/eclipse/eclipse-5.zip)](https://s3.amazonaws.com/caelum-online-public/eclipse/eclipse-5.zip), **ele será necessário para a realização dos exercícios.**

Refatorando o Importador

A classe `ImportadorDeGastos` é capaz de criar uma lista de gastos a partir de um arquivo escrito num formato determinado. Porém, por causa de um bug na aplicação que gera os arquivos, alguns deles possuem gastos repetidos. Rodando a classe `TesteImportador` vemos que o mesmo gasto aparece 3 vezes.

Para resolver isso, em vez de retornar `List` no método `importa` , vamos mudar o código para retornar um `Set` , que por natureza não permite objetos repetidos. Para ver se é seguro fazer isso, vejamos os usos do método `importa` para garantir que ninguém depende do retorno ser especificamente uma `List` . Para isso use o **ctrl + alt + H** em cima do nome do método e navegue pelos usos.

Perceba que nenhum dos usos usa métodos específicos de `List` , então a primeira coisa que vamos fazer é trocar por `Collection` . Voltando aos usos do método, usemos o **ctrl + 1** para mudar o tipo das variáveis para `Collection` .

Agora, no método `importa` podemos mudar o tipo da variável gastos para `Collection` . Tudo compila ainda. Agora como queremos evitar repetições, vamos trocar a implementação de `ArrayList` para algum `Set` . A implementação mais usada é o `HashSet` , mas ele não garante ordem, então, como ordem é importante para nós, vamos usar um `LinkedHashSet` .

```
Collection<Gasto> gastos = new LinkedHashSet<Gasto>();
```

Perceba que ao usar `new Lin<ctrl + espaço>` , o `LinkedHashSet` nem aparece nas primeiras opções. Para otimizar o **`<ctrl + espaço>`** , podemos usar a estratégia de colocar partes do nome, respeitando as maiúsculas: `new LinHaS<ctrl + espaço` ou, simplesmente, as iniciais: `new LHS<ctrl+espaço>` .

Vamos rodar novamente a classe `TestaImportador` e ver o resultado. Ainda são mostrados 3 gastos iguais, mas por quê?

O `Set` decide se um objeto é igual a outro pelo método `equals` , e não sobrescrevemos esse método na classe `Gasto` . A implementação padrão dele compara as referências dos objetos. Então, mesmo que os dados sejam os mesmos, são objetos diferentes na memória, logo o `equals` padrão vai retornar `false` .

Para o `Set` funcionar do jeito que queremos, precisamos definir qual é o nosso critério para dizer se dois objetos são iguais, ou seja, sobrescrever o método `equals` . Porém, todas as classes baseadas em *hashes* comparam os `hashCode` s dos objetos antes de comparar os objetos em si - trocando uma operação custosa por uma barata e melhorando a eficiência. Leia mais sobre esse assunto na apostila do [CS-14, disponível para download (http://www.caelum.com.br/apostilas/)](http://www.caelum.com.br/apostilas/) no site da Caelum.

Em resumo, para que o `LinkedHashSet` faça seu trabalho, além de sobrescrever o `equals` , precisamos sobrescrever o método `hashCode` de forma que objetos iguais sempre retornem o mesmo `hashCode` - conforme o contrato estabelecido no *Javadoc* desses dois métodos.

Gerar esses dois métodos baseados nos dados da classe `Gasto` parece meio complicado, não? Para nossa sorte, o Eclipse já sabe gerar esses métodos para nós, já seguindo o contrato de consistência descrito no *Javadoc*, então não precisamos nos preocupar com os detalhes técnicos. Só precisamos falar quais são os campos da classe que definem se dois objetos são iguais. Esse gerador se chama *Generate hashCode() and equals()*, então conseguimos acessá-lo via **ctrl+3** e uma parte do nome, por exemplo, **ctrl+3 equals**.

Na tela que aparece, vamos selecionar todos os atributos e apertar **enter**. Repare que o *wizard* vai te dar um *warning*, já que a classe `Funcionario` também deveria implementar o `equals` e o `hashCode` corretamente. Confirmando, conseguimos ver os métodos gerados. Note a complexidade deles - ainda bem que não tivemos que fazer isso na mão. Veja que eles delegam para os equals e hashCode dos outros objetos, supondo que estão implementados da forma certa - por isso o *warning* anterior.

Devemos, então, fazer a mesma coisa na classe `Funcionario` , senão o que fizemos na classe `Gasto` pode nos dar resultados incorretos.

Agora, ao rodar o teste, apenas um gasto é impresso. Essa impressão usa o `toString()` que implementamos em um dos capítulos anteriores, mas se quisermos mudar esse método para imprimir mais dados o trabalho começa a ficar grande. O Eclipse também nos ajuda a implementar o `toString` de uma forma fácil, usando o menu ***Generate toString()***.

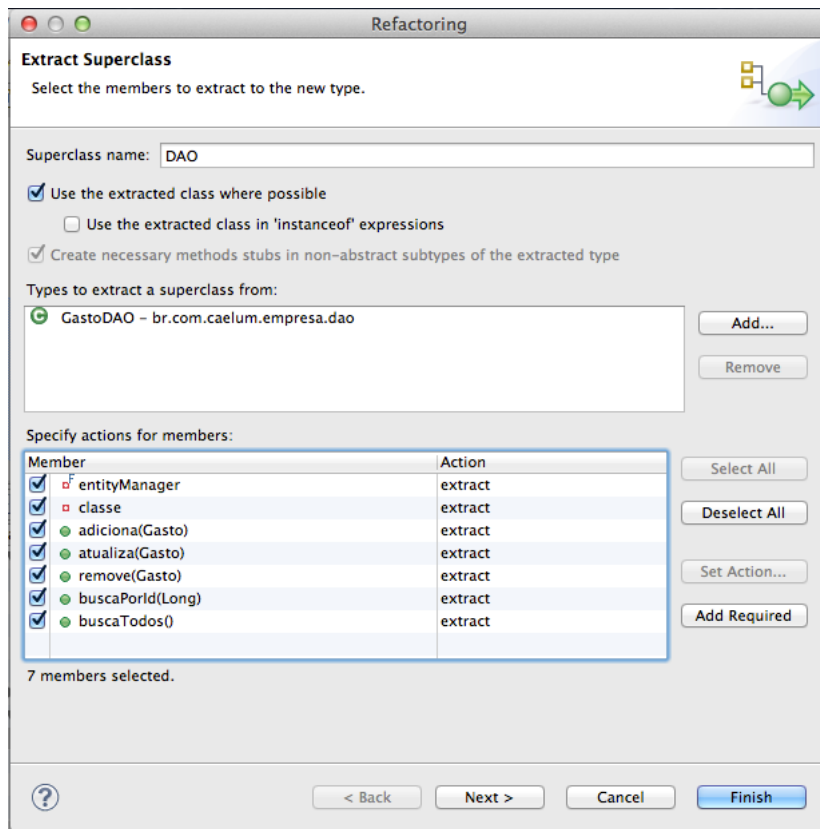
Acessando esse menu com a ajuda do **ctrl + 3**, veremos uma tela em que podemos personalizar o texto de várias maneiras, como selecionar atributos, mudar ordem, usar concatenação com *StringBuffer*, etc.

Agora que temos os gastos sendo importados corretamente, podemos gravar estas informações no banco de dados. Para isto, vamos utilizar a classe `GastoDAO`. Ela tem todas as funcionalidades mais básicas para interagirmos com o banco de dados, como adicionar, atualizar, remover e listar gastos. Vejamos o código dessa classe, que está utilizando `JPA` como mecanismo de persistência:

```
public class GastoDAO {

    private final EntityManager entityManager = new JPAUtil().getEntityManager();
    private Class<Gasto> classe = Gasto.class;

    public void adiciona(Gasto entity) {
        entityManager.persist(entity);
    }

    public Gasto atualiza(Gasto entity) {
        return entityManager.merge(entity);
    }

    public void remove(Gasto entity) {
        entityManager.remove(entity);
    }

    public Gasto buscaPorId(Long id) {
        return entityManager.find(classe, id);
    }

    public List<Gasto> buscaTodos() {
        Query query = entityManager
                .createQuery("from " + classe.getName());
        return query.getResultList();
    }

}
```

Também precisaremos gerenciar os funcionários em nossa aplicação, pois para incluir o gasto, precisamos incluir o funcionário antes. Logo, vamos ter que criar uma classe `FuncionarioDAO`, com praticamente o mesmo código existente em `GastoDAO`. Como já vimos antes, não devemos repetir código, então vamos alterar nosso `GastoDAO`, extraindo dele toda parte que é comum e que pode ser reutilizada em uma nova classe.

O primeiro passo será extrairmos nosso código comum para uma superclasse, que será utilizada como base para nossos DAOs, contendo todo código comum. Utilize o atalho **alt + shift + T** para exibir o menu de refactor; agora selecione *Extract Superclass...* para extrair a superclasse. Na janela que abriu, podemos selecionar o nome de nossa classe mãe, que será `DAO`, além de quais membros vamos mover. Vamos selecionar todos os métodos e atributos clicando em *Select all*, e em *Finish* para criarmos a classe `DAO`.



Ao final, teremos a nova classe `DAO` :

```
public class DAO {

    private final EntityManager entityManager = new JPAUtil().getEntityManager();
    private Class<Gasto> classe = Gasto.class;

    public DAO() {
        super();
    }

    public void adiciona(Gasto entity) {
        entityManager.persist(entity);
    }

    public Gasto atualiza(Gasto entity) {
        return entityManager.merge(entity);
    }

    public void remove(Gasto entity) {
        entityManager.remove(entity);
    }

    public Gasto buscaPorId(Long id) {
        return entityManager.find(classe, id);
    }

    public List<Gasto> buscaTodos() {
        Query query = entityManager
                .createQuery("from " + classe.getName());
        return query.getResultList();
    }
}
```

E a classe `GastoDAO` , que agora tem este conteúdo:

```
public class GastoDAO extends DAO {

}
```

Teremos que fazer algumas alterações na classe `DAO` , já que ela só consegue inserir objetos do tipo `Gasto` ainda. Temos que deixá-la mais abstrata a ponto de conseguir reutilizar seu código para qualquer objeto que quisermos inserir no banco de dados.

A forma mais eficiente de fazermos isso é utilizando o recurso *Generics*, introduzido no Java 5. Vamos transformar nossa classe `DAO` em um [DAO genérico (http://blog.caelum.com.br/ei-como-e-o-seu-dao-ele-e-tao-abstraido-quanto-o-meu/)](http://blog.caelum.com.br/ei-como-e-o-seu-dao-ele-e-tao-abstraido-quanto-o-meu/), para facilitar a reutilização.

Infelizmente, não existe um atalho para transformar uma classe em genérica de maneira simples. Teremos que realizar esta alteração manualmente, mas, mesmo assim, vamos utilizar os atalhos vistos anteriormente para facilitar esse trabalho extra.

Inicialmente, vamos alterar a declaração da classe para que ela receba um tipo genérico **T**. O código resultante será o seguinte:

```
public class DAO<T>{
//... resto do código
```

Agora, vamos trocar todas as referências a `Gasto` para referências ao tipo **T**. Não temos um atalho para isso, então vamos selecionar e substituir o termo **Gasto** por **T**, usando **ctrl + F**.

Após organizarmos os imports (**ctrl + shift + O**), temos apenas um ponto não compilando em nossa classe, nosso atributo `classe`, usado pelo `EntityManager` para realizar as buscas dos objetos. [Não conseguimos descobrir a classe de tipos genéricos (http://blog.caelum.com.br/nao-posso-descobrir-nem-instanciar-tipos-genericos-porque/)](http://blog.caelum.com.br/nao-posso-descobrir-nem-instanciar-tipos-genericos-porque/), por isso não podemos fazer `T.class`. Não temos como saber qual será o tipo real de **T** aqui, então vamos pedir para que quem for utilizar o `DAO` informe qual a classe real do objeto, passando essa informação no construtor de nossa classe.

Utilizando o **ctrl + 3** e escolhendo *Generate constructor using fields*, teremos o seguinte código no final:

```
public class DAO<T> {

    private final EntityManager entityManager = new JPAUtil()
            .getEntityManager();
    private Class<T> classe;

    public DAO(Class<T> classe) {
        super();
        this.classe = classe;
    }

    public void adiciona(T entity) {
        entityManager.persist(entity);
    }

    public T atualiza(T entity) {
        return entityManager.merge(entity);
    }

    public void remove(T entity) {
        entityManager.remove(entity);
    }

    public T buscaPorId(Long id) {

        return entityManager.find(classe, id);
    }

    public List<T> buscaTodos() {
        Query query = entityManager.createQuery("from " + classe.getName());
        return query.getResultList();
    }
}
```

Opa, mas agora a classe `GastoDAO` parou de compilar, o que aconteceu? Como criamos um construtor que recebe um parâmetro, agora temos que passá-lo na hora de criar nosso `GastoDAO`. Vamos gerar esse construtor usando o **ctrl + 1**,e também vamos colocar o tipo genérico do `DAO`. O resultado final será:

```
public class GastoDAO extends DAO<Gasto> {

    public GastoDAO() {
        super(Gasto.class);
    }

}
```

Nosso DAO agora está pronto!