02

Configurando o ambiente

Para tirar máximo proveito do Eclipse, podemos usar diversas configurações específicas que dão maior feedback, facilitam a digitação, ou ainda melhoram os outros atalhos e views. Essas configurações vão das mais complexas às mais simples, como, por exemplo, mostrar os números das linhas nos arquivos. Isso é bem útil quando estamos passando por uma stacktrace ou quando queremos ter uma idéia melhor do tamanho de uma classe ou método. Para habilitar essa configuração você pode clicar com o botão direito do mouse na barrinha acinzentada do lado esquerdo do editor e selecionar Show line numbers. Alternativamente, podemos usar o ctrl + 3.

Na nossa aplicação, poderíamos, seguindo as idéias do capítulo anterior, criar mais um teste que verifica que se a lista não tem gastos acima do limite, a lista devolvida é vazia. Nosso teste também ficaria na classe `FiltradorDeGastosEspeciaisTest` e teria como implementação o seguinte código:

```
@Test
public void devolveListaVaziaQuandoNaoHaGastosAcimaDoLimite() throws Exception {
    Funcionario funcionario = new Funcionario("Vitor", 107, new GregorianCalendar(1989, 11, 28));
    Calendar hoje =
        Calendar.getInstance();

    Gasto g1 = new Gasto(14.0, "almoco", funcionario, hoje);
    Gasto g2 = new Gasto(22.0, "jantar", funcionario, hoje);
    List<Gasto> gastos = Arrays.asList(g1, g2);

    FiltradorDeGastosEspeciais filtrador = new FiltradorDeGastosEspeciais(25.0);
    List<Gasto> lista = filtrador.filtraGastosGrandes(gastos);
              assertEquals(0, lista.size());
}
```

Esse teste passa, mas o código está todo mal indentado e tem espaços em lugares estranhos. Felizmente, sabemos o atalho que resolve esses problemas: o ctrl + shift + F cuida de formatar o código para nós. Quando usamos esse comando, no entanto, criamos um novo problema: a configuração padrão do Eclipse quebra a linha no octagésimo caractere! Isso acontece porque, para resoluções baixas, esse era um bom número. Hoje, pode não ser uma boa idéia.

Para resolver esse problema, temos a configuração do Formatter, que define as regras para a formatação automática. Para isso, vá em Window > Preferences e digite Formatter, selecionando o que está em Java > Code Style > Formatter, ou use o ctrl + 3 da maneira usual. Temos alguns perfis de formatação disponíveis e podemos criar um novo para personalizá-lo.

Ao clicar em New e dar um nome para o perfil, temos dezenas de opções e o Preview ao lado que ilustra os efeitos da opção modificada. Temos opções para indentação, posição das chaves (Braces), espaços em branco, linhas, formato do `for, if, while` e várias outras. Nosso problema aborda uma das configurações de formatação mais fundamentais: precisamos aumentar o tamanho máximo de uma linha. Em Line Wrapping > Maximum line width, basta alterar o padrão para, digamos, 120 caracteres.

Outro problema frequente é o de código sujo indo para o controle de versão (http://www.caelum.com.br/curso/online/git/). Se você trabalha em equipe, é muito provável que você use um sistema de controle de versão e, assim, é interessante que não haja imports nem variáveis não utilizadas perdidas nas alterações. Também há times que preferem ter a anotação `@Override` em toda sobrescrita de método - para evitar sobrecargas acidentais em refatorações, por exemplo.

Como boa prática, então, roda-se o Clean Up do projeto (ctrl + 3 Clean Up) antes de fazer o commit e o configuramos para que faça tais verificações para nós. A configuração do Clean Up (também em Java > Code Style) define algumas ações relacionadas à limpeza do código. Ao clicar em New e nomear esse conjunto de preferências, definimos nossas próprias regras. Com esse comando configurado basta apenas lembrarmos de executá-lo sempre antes de fazer o commit.

Mas... se precisamos lembrar disso, é bem provável que, hora ou outra, alguém esqueça de fazê-lo e suje o projeto. Para não depender de lembrar de fazer o Clean Up podemos ir mais além e mandar o Eclipse fazer isso automaticamente, sempre que salvarmos um arquivo. Esse é o recurso chamado Save Actions (em Java > Editor). Podemos habilitá-lo, selecionando o que será executado toda vez que um arquivo for salvo, como formatar o arquivo, organizar os imports e fazer o Clean Up (additional actions).

É muito importante que todos da mesma equipe (ou da mesma empresa dependendo do caso) usem as mesmas configurações de formatação e de Clean Up, para evitar que os arquivos fiquem sendo alterados constantemente por conta de diferenças nas convenções e configurações usadas. Para atingir isso, é possível exportar essas configurações e importá-las em outras máquinas. É possível, também, mudar essas configurações por projeto, assim cada um se comporta da maneira convencionada.

Outra coisa que incomoda muitos desenvolvedores são as marcações de erros e warnings em situações que gostaríamos de ignorá-los. Por exemplo, quando criamos uma exceção nossa, extendemos a classe `Exception` (ou uma de suas filhas) e essa, por sua vez, é uma classe serializável. A verificação que marca com warning todas as classes que são serializáveis e que não têm Serial Version UID pode ser um tanto chata quando não pretendemos fazer uso desse recurso.

O Eclipse permite que nós suprimamos esse aviso, assim como diversos outros, nas preferências de Errors/Warnings em Java/Compiler - para acessá-lo, use o ctrl + 3.

Por outro lado, há casos que realmente gostaríamos que certos problemas dêem erros de compilação "artificiais": se não queremos que, de forma alguma, usemos código depreciado no desenvolvimento do nosso projeto, podemos marcar os itens de Deprecated and restricted API como error .

Esses itens passarão a aparecer na view Problems, também, para que você localize de forma eficiente todas essas situações. Manter a lista de problemas sempre vazia é muito importante para nos dar alguma garantia de que o projeto continua consistente com nossas convenções.

Trabalhando com o Eclipse da forma que indicamos até agora, abusando de ctrl + espaço e de ctrl + 1, é frequente que terminemos de escrever o código sem estar no fim da linha. Isso acontece, por exemplo, quando preenchemos a condição de um if: terminamos de escrever a condição e nos vemos forçados a levar o cursor até o final da linha para abrir chaves.

Se quisermos melhorar ainda mais a ajuda que o Eclipse nos dá enquanto estamos digitando, podemos ir à configuração Typing (em Java > Editor) e habilitar a inserção de chaves e ponto-e-vírgulas nos lugares certos (Braces e Semicolons). Com esses checkboxes marcados, ainda que estivermos no meio da linha basta digitarmos ";" ou "{" esse caractere será inserido no final da linha. Outros exemplos de atividades que se beneficiam dessa configuração são quando estamos escrevendo uma `String` ou durante a passagem de parâmetros para um método.

Pra quem gosta de copiar uma mesma linha várias vezes e mudar apenas o necessário, é comum que o comportamento padrão do ctrl + espaço atrapalhe muito! Se estivermos implementando um método que mostra um funcionario e ele deva imprimir a matrícula e o nome do funcionário, por exemplo, podemos nos basear na linha de impressão da matrícula para fazer a outra. Em determinado momento, teríamos algo como:

```
public void mostra() {
    System.out.println(this.getMatricula());
    System.out.println(this.getMatricula());
}
```

Contudo, se posicionarmos o cursor na posição `get|Matricula`, apertarmos ctrl + espaço e escolhermos o método `getNome()`, ficaremos com algo estranho como:

```
public void mostra() {
    System.out.println(this.getMatricula());
    System.out.println(this.getNome()Matricula());
}
```

Para que o Eclipse opte por mudar seu código em vez de simplesmente adicionar mais código é possível configurar o Content Assist (o nome oficial do atalho ctrl + espaço). É só, nesse menu, trocar Completion inserts por Completion Overwrites que muda o jeito que o código completado se comporta. Por exemplo se estamos usando `this.getMatricula()` com o cursor depois do `get` e usarmos o ctrl + espaço para trocar para `this.getNome()`. Habilitando a sobrescrita, o resultado da operação anterior será:

```
public void mostra() {
    System.out.println(this.getMatricula());
    System.out.println(this.getNome());
}
```

Outra configuração interessante nessa mesma tela é a decisão entre Insert parameter names e Insert best guessed arguments. Suponha que o construtor da classe Funcionario está assim:

```
public Funcionario(String nome, int matricula, Calendar dataNascimento) {...}
```

Quando já tivermos as variáveis com os valores que serão colocados no funcionario, ao dar `new Funcionario()` usando o ctrl + espaço a primeira opção dos argumentos do construtor (ou de um método qualquer) serão:

```
String joao = "João";
int treze = 13;
Calendar agora = Calendar.getInstance();

new Funcionario(nome, matricula, dataNascimento); // Insert parameter names
```

Quer dizer, o Eclipse assume que suas variáveis locais terão o mesmo nome dos parâmetros do construtor. Há, contudo, uma forma de pedir que o Eclipse tente adivinhar quais variáveis vão em cada posição do construtor. O chute não será certeiro todas as vezes, mas pode ser uma boa ajuda em vários casos! Com a configuração de Insert best guessed arguments a saída será a seguinte:

```
String joao = "João";
int treze = 13;
Calendar agora = Calendar.getInstance();

new Funcionario(joao, treze, agora) // Insert best guessed arguments
```

Em todo caso você pode alternar entre os parâmetros com o tab e selecionar o valor que mais se aproxima do que você quer.

Outro problema que assola desenvolvedores no dia-a-dia são as muitas possibilidades ao fazer o import de classes. Quem de nós nunca importou o `java.awt.List` em vez do `java.util.List` ? Se nunca utilizaremos as classes do pacote `java.awt` ou do `org.corba` não gostaríamos nem que eles fossem opções elegíveis na hora de importar.

Ainda na mesma tela de configuração existe um link para type filters. Se clicarmos nele, podemos adicionar classes e pacotes que nunca serão sugeridos no ctrl + espaço nem no ctrl + shift + T. Por exemplo, excluir os pacotes `java.awt.*` e `org.corba.*` .

Algumas classes de bibliotecas que usamos foram feitas para agrupar diversos métodos estáticos utilitários. É o caso da classe `Math` do Java, com funções matemáticas, `Collections` e `Arrays` , com operações úteis de coleções ou arrays e `Assert` do JUnit, com métodos para fazer asserções. Para garantir que dois objetos são iguais, no JUnit, podemos fazer:

```
Assert.assertEquals(objeto, outroObjeto);
```

Se quisermos diminuir o ruido do código, podemos fazer o import estático desse método, com o cursor no método e ctrl + shift + M, resultando em:

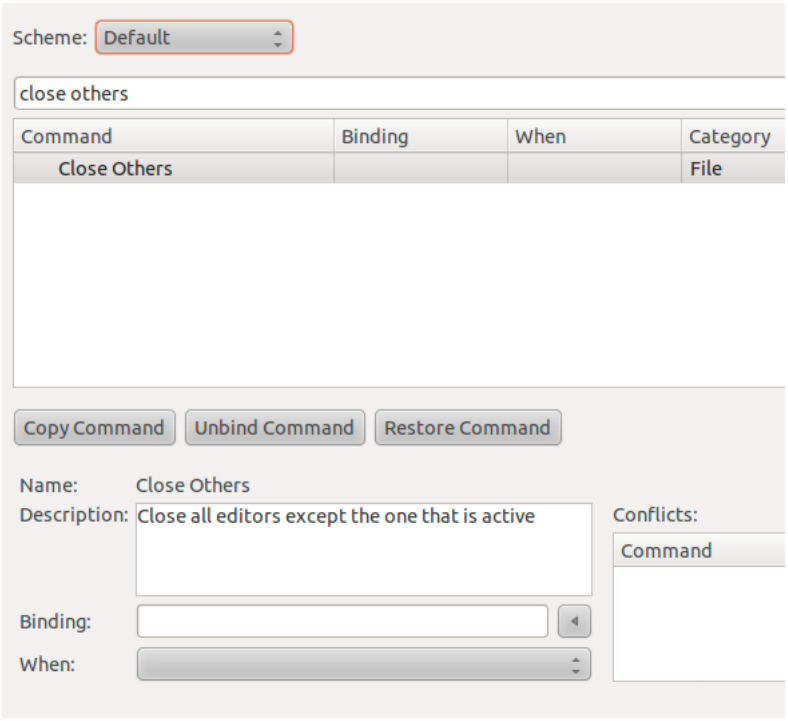
```
import static org.junit.Assert.assertEquals;

//... e, quando for usa-lo
assertEquals(objeto, outroObjeto);
```

Mesmo assim precisamos digitar o nome da classe `Assert` para poder chamar o método. Contudo podemos ensinar o eclipse a importar os métodos dessa classe (e de qualquer outra) automaticamente. Isso está na configuração Favorites (use o ctrl + 3), onde você pode adicionar classes cujos métodos estáticos sempre vão fazer parte das sugestões do ctrl + espaço. Por exemplo se adicionarmos o tipo `org.junit.Assert` (em New Type), podemos fazer em qualquer classe do projeto (se o JUnit estiver no classpath) `assert` e selecionar um dos métodos, que será importado estaticamente e já inserido no código.

Finalmente, é muito importante usar atalhos para as operações que você mais usa. Assim economizamos o tempo de ir até o menu, ou mesmo de digitar ctrl + 3 e começar a digitar o nome da ação. Por exemplo, se você costuma renomear elementos com frequência, usar o ctrl + 1 e as setas para escolher a opção de renomear (ou ir ao menu) começa a tomar muito tempo. Usar o atalho direto pra isso (alt + shift + R) é muito mais rápido e direto. Para ver uma lista de todos atalhos disponíveis, podemos usar o ctrl + shift + L, assim podemos descobrir qual é o atalho para a operação que desejamos.

Entretanto, nem todas as operações possuem atalhos associados ou, ainda, possuem um atalho que conflita com algum do sistema operacional. Por exemplo temos um atalho para fechar um arquivo aberto no editor - ctrl + W, outro para fechar todos os arquivos abertos - ctrl + shift + W, mas não temos um atalho para fechar todos os arquivos abertos, exceto o que estamos editando agora - só conseguimos fazer isso com botão direito na aba e Close Others ou com a ajuda do ctrl + 3. Então vamos criar um atalho pra isso, digitando ctrl + 3 Keys, que abrirá uma tela com todas as ações possíveis e os atalhos associados. Podemos filtrar os atalhos, digitando Close Others na caixa de texto e selecionando a ação.



No campo Binding podemos digitar o atalho desejado, por exemplo ctrl + alt + W. Em Conflicts podemos ver se esse atalho já está sendo usado em outra ação. Em caso afirmativo, então devemos escolher outro ou remover o atalho da ação que apareceu. Podemos escolher outro, por exemplo ctrl + alt + shift + W, que não está associado a nenhuma ação, e confirmar. Agora é só usar esse novo atalho quando quisermos fechar os outros editores.