09

Para saber mais: Por que usamos a palavra-chave 'final' no parâmetro do método 'enviar'?

Repare no método `enviar` da classe `ChatService`:

```
public class ChatService {
    public void enviar(Mensagem mensagem) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                String texto = mensagem.getTexto();

                 // Restante do código omitido...

            }
        });
        ...
}
```

Ao tentarmos rodar o código dessa forma, receberemos um erro de compilação ao tentarmos acessar o texto da mensagem fazendo `mensagem.getTexto()`. Isso ocorre porque, como você já sabe, não sabemos quando uma `Thread` irá executar. Por isso, pode haver o risco de a Thread ser executada muito depois do método `enviar(mensagem)` ter finalizado sua execução, levando com ele todas as variáveis de seu escopo. Em nosso código, gostaríamos de poder acessar a variável `mensagem` de dentro da `Thread`, o problema é que ela tem seu próprio escopo e só podemos acessar variáveis desse escopo.

O Java não possui suporte total a *Closures*, ou seja, até podemos acessar variáveis de outro escopo mas não podemos alterar o seu valor. Isso acontece porque quando acessamos uma variável de outro escopo (da classe externa, por exemplo) o compilador precisa criar uma cópia dessa variável externa para o escopo local. Para não correr o risco de ocorrer uma inconsistência no valor das duas variáveis (a externa e a cópia), é necessário o uso da palavra-chave `final`.

Vamos comprovar isso através de um experimento?

1) Crie uma pasta em algum lugar em seu computador para colocarmos os arquivos.

2) Dentro dessa pasta, crie o arquivo `TestandoClosure.java` com o código abaixo:

```
import java.lang.reflect.Field;

public class TestandoClosure {
    public static void main(String[] args) {
        final String[] alunos = {"Leonardo", "Nico", "Rômulo"};
        new Runnable() {
            public void run() {
                for (String aluno : alunos)
                    System.out.print(aluno);
            }
        }.run();
    }
}
```

Nessa classe estamos criando uma classe anônima de `Runnable`.

4) Abra o terminal (se você usa o Windows, pode fazer no Prompt) e compile essa classe `TestandoClosure`.

```
javac TestandoClosure.java
```

Após isso teremos dois arquivos `.class` (lembrando que classe anônima é uma classe gerada em tempo de compilação!)

```
MacBook-Pro-de-Leonardo:teste leonardocordeiro$ ls
TestandoClosure$1.class TestandoClosure.class  TestandoClosure.java
```

5) Vamos usar a ferramenta `javap` para visualizar o *bytecode* da classe anônima:

No terminal, digite: `javap TestandoClosure\$1.class`

```
MacBook-Pro-de-Leonardo:teste leonardocordeiro$ javap TestandoClosure$\1.class
Compiled from "TestandoClosure.java"
final class TestandoClosure$1 implements java.lang.Runnable {
  final java.lang.String[] val$alunos;
  TestandoClosure$1(java.lang.String[]);
  public void run();
}
```

Repara que ele cria um atributo chamado `val$alunos` que é uma cópia para o array `alunos`. Além disso há um construtor que recebe um `String[]` para associar ao atributo. Podemos ir mais além e ver essa associação no bytecode.

Para isso, no terminal, digite: `javap -v TestandoClosure\$1.class` (Agora com a opção `-v`. É necessário escapar o caracter `$`, por isso usamos o `$`).

```
TestandoClosure$1(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags:
  Code:
    stack=2, locals=2, args_size=2
       0: aload_0
       1: aload_1
       2: putfield      #1                  // Field val$alunos:[Ljava/lang/String;
       5: aload_0
       6: invokespecial #2                  // Method java/lang/Object."<init>":()V
       9: return
    LineNumberTable:
      line 9: 0
```

Esse é trecho do construtor da classe anônima, onde podemos ver uma associação (`putfield`) ao atributo `val$alunos`.