Favoreça imutabilidade e simplicidade

Quando começamos a programar com Java, passamos por um código semelhante ao que segue:

```
String s = "java";
s.toUpperCase();
System.out.println(s);
```

O código acima, clássico nas provas de certificação, imprimirá a `String` em letras minúsculas. Isso porque, como sabemos, **Strings são imutáveis**. Todo método que você invoca em uma `String` que parece modificá-la, na verdade, faz o Java instanciar um objeto novo, sem alterar o original, e devolver uma referência àquele.

Mas porque Strings são imutáveis? Porque escrever objetos imutáveis? Quais as vantagens? E porque muitas de nossas classes deveriam ser imutáveis? Veremos que, ao tornar uma classe imutável, uma ampla gama de problemas comuns desaparecem [23, 11].

Simplicidade e previsibilidade

O primeiro ponto é que objetos imutáveis são muito mais simples de se usar que objetos mutáveis, pois acabam tendo um comportamento mais que previsível:

```
String s = "arquitetura";
metodoMisterioso(s);
System.out.println(s);
```

O que será mostrado na terceira linha, com absoluta certeza? Não importando o que o `metodoMisterioso` faça, aparecerá *arquitetura*. **Objetos imutáveis não sofrem efeitos colaterais, pois têm comportamento previsível em relação ao seu estado**. Compare com um código semelhante mas passando uma referência a um objeto mutável, no caso um `Calendar`:

```
Calendar c = Calendar.getInstance();
metodoMisterioso(c);
System.out.println(c.get(Calendar.YEAR));
```

Qual ano será impresso na terceira linha acima? O ano atual? Não é possível saber só com essas informações, pois o `metodoMisterioso` pode ter alterado o ano do nosso objeto `Calendar`.

Os próprios engenheiros da Sun já admitiram alguns erros de design das APIs antigas, como a classe `java.util.Calendar` ser mutável [16]. Por isso, muitas vezes recorremos às APIs de terceiros como a **Joda Time**, onde encontramos entidades de datas, horários e intervalos imutáveis.

Quando o objeto é mutável, para evitar quem alguém o modifique, precisamos sempre interfaceá-lo ou embrulhá-lo de tal forma que os métodos expostos não possibilitem modificação. É como, por exemplo, fazemos frequentemente com coleções através do `Collections.unmodifiableList(List)` e seus similares, criando uma **cópia defensiva**:

em vez de passar a referência original adiante, com medo de ocorrer uma modificação por parte do código de outra pessoal.

Objetos imutáveis são mais simples de se lidar. Depois de sua criação, sempre saberemos seu valor e não precisamos tomar cuidados adicionais para preservá-lo. Já objetos mutáveis podem ser mudados e, frequentemente, temos que passar cópias dos mesmos com o objetivo de evitar alterações indevidas.

Otimizações de memória

Podemos tirar proveito da imutabilidade de objetos de outras formas [22]. Como cada objeto representa apenas um estado, você não precisa mais do que uma instância de cada estado. A própria API da linguagem Java usa a ideia de imutabilidade como uma vantagem para fazer cache e reaproveitamento de objetos.

Existe, dentro da VM, um *pool* de Strings que faz com que Strings com o mesmo conteúdo na verdade possam ser representadas com um mesmo objeto compartilhado (no caso, a array privada de caracteres pode ser compartilhada). O mesmo acontece em boa parte das implementações das classes wrapper como `Integer`. Ao invocar `Integer.valueOf(int)`, o objeto `Integer` devolvido pode ser fruto de um cache interno de objetos com números frequentemente solicitados [39].

Esse tipo de otimização só é possível com objetos imutáveis. É seguro compartilhar instâncias de `String`s ou `Integer`s com várias partes do programa, sem o medo de que alguém possa alterar o objeto e afetar os outros.

E há mais possibilidades ainda para otimizações graças à imutabilidade. A classe `String`, por exemplo, ainda se aproveita de sua imutabilidade para compartilhar seu array de `char` interno entre `String`s diferentes. Se olharmos o código fonte da classe `String`, veremos que ela possui três atributos principais: um `char[] value` e dois inteiros, `count` e `offset` [36]. Os dois inteiros servem para controlar o início e o fim da String dentro do array de char.

O seu método `substring(int,int)` leva em conta a imutabilidade das `String`s e os dois inteiros que controlam início e fim para reaproveitar o array de `char`. Quando pedimos uma determinada *substring*, ao invés do Java criar um novo array de `char` com o pedaço em questão, ele devolve um novo objeto `String` que internamente possui uma referência para o mesmo array de char que a `String` original, e tem apenas os seus dois índices inteiros ajustados. Ou seja, criar uma substring em Java praticamente não gasta mais memória, sendo apenas uma questão de ajustar dois números inteiros

Essa otimização é uma implementação um pouco mais simples do design pattern *Flyweight*, onde propõe-se reaproveitar o estado interno entre objetos imutáveis com objetivo de reaproveitar o uso memória [**?**]. Poderíamos até ir mais longe com o pattern *flyweight*, por exemplo construindo `String`s a partir de mais de uma `String` já usada em memória (nesse caso a API do Java prefere não chegar a esse ponto, pois apesar de um ganho de memória, haveria o *tradeoff* de perda de performance).

É válido ressaltar também o perigo dessa estratégia de otimização. Uma `String` pequena pode acabar segurando referência para uma array de `char`s muito grande se ela foi originada a partir de uma String grande, impedindo que essa array maior seja coletada mesmo se não possuirmos referências para a `String` original. Outro problema comum de aparecer com objetos imutáveis é o excesso de memória consumido tempora-

riamente. Cada invocação de método pode criar uma nova cópia do objeto apenas com alguma alteração e, se isto estiver dentro de um laço, diversas cópias temporárias serão utilizadas, mas apenas o resultado final é guardado. Aqui, o uso do design pattern *builder* pode ajudar, como é o caso de usarmos a classe mutável `StringBuilder` (e sua versão thread safe `StringBuffer`) e seu método `append` em vez do `String.concat` (ou o operador sobrecarregado `+`) em um laço.

Objetos imutáveis trazem uma segurança tão grande em relação a compartilhar instâncias sem se preocupar com alterações indevidas que podemos usar esse fato a nosso favor quando for importante economizar memória.

Acesso por várias threads

Com um objeto mutável é frequente encontrar um cenário onde duas threads o compartilham e, em dado momento, alguma delas realiza uma mudança em seu estado. A mudança em si não é um problemas, mas o intervalo entre o antes e o depois da mudança sim. Nesse intervalo, o objeto pode estar em um estado considerado inconsistente onde não está apto a receber outra invocação de método sem que alguma informação seja perdida ou até duplicada.

Imagine uma classe `Conta` com operações de manipulação do `saldo` como a abaixo:

```
public class Conta {
    private double saldo;

    public void deposita(double valor) {
        this.saldo += valor;
    }
}
```

O que acontece se duas thread tentarem, simultaneamente, depositar valores em um mesmo objeto `Conta`? Se uma thread tenta depositar 100 reais e outra, ao mesmo tempo, tenta depositar 200 reais, no fim, teremos mais 300 reais?

```
Thread #1:          |   Thread #2:
                    |
c.deposita(100);    |   c.deposita(200);
```

Como sabemos, operações concorrentes podem ter problemas ao alterar os mesmos dados simultaneamente. O problema aqui é que as alterações não são atômicas e podem ser interrompidas no meio. A simples linha `this.saldo += valor`, quando compilada, originará seis bytecodes diferentes que trarão muitas possibilidades de entrelaçamento para o escalonador. Imagine, por exemplo, a seguinte sequência de execução possível no cenário acima:

```
- Thread #1: Recupera o valor do atributo (0)
- Thread #2: Recupera o valor do atributo (0)
- Thread #1: Faz a soma do valor 0 com o valor 100 (resultado = 100)
- Thread #2: Faz a soma do valor 0 com o valor 200 (resultado = 200)
```

```
- Thread #1: Atribui o resultado (100) de volta ao atributo
- Thread #2: Atribui o resultado (200) de volta ao atributo
```

No final, o atributo acabaria valendo 200 e não os 300 esperados. Os problemas de concorrência surgem nessa hora, quando uma thread lê o valor do objeto e ele ainda está em um estado intermediário, na transição de um estado consistente para outro.

Nesse sentido, podemos recorrer a algumas funcionalidades da linguagem como blocos `synchronized` no caso de Java, ou as utilidades do JUC (pacote `java.util.concurrent`). Seja qual for a abordagem, **tratar problemas de concorrência é extremamente difícil e complicado de detectar erros**, em especial porque alguns bugs surgem muito raramente, quando condições específicas de entrelaçamento de execução de duas ou mais threads ocorrerem.

Em vez de recorrer aos recursos da linguagens, vamos pensar nesses objetos de forma diferente. Se o problema é esse estado intermediário, elimine-o! Não deixe o objeto mudar de valor. Essa é a idéia da **imutabilidade**: uma vez que o objeto foi criado, ele nunca muda, sempre está com o estado original e consistente.

A vantagem mais clara aqui é o ganho de *thread-safety*, pois como não existem estados intermediários, não há como acessar/buscar/modificar dados em momentos de inconsistência. O estado inconsistente fica escondida na lógica de construção do objeto e o objeto novo só deverá ser acessado quando terminar de ser construído, para então ter sua referência compartilhada por mais de uma thread.

As linguagens funcionais mais puras, como *Erlang*, estão muito faladas em ambientes de grande concorrência dada sua característica de trabalhar quase que apenas com valores imutáveis. Perde-se o conceito de *variável* como o conhecemos, já que os valores não mudam: nascem e morrem com o mesmo estado. Você é obrigado a programar apenas de maneira imutável, o que é uma enorme mudança de paradigma.

Imutabilidade e interfaces fluentes

Há ainda outro pattern que aparece com frequencia na presença de objetos imutáveis: o *fluent interface* [18]. Já que todos os métodos não modificam o objeto em questão, a única possibilidade de algo parecido com uma alteração é criar uma cópia do objeto, e essa cópia sim possui a modificação. É comum que, para isso, alguns métodos desta classe devolvam o novo objeto modificado, possibilitando a concatenação de invocação de métodos, como também ocorre com a classe `String`:

```
String valor = "  arquiteturajava.com.br "
String resultado = valor.toUpperCase().trim().substring(6);
```

O mesmo ocorre com algumas outras classes imutáveis muito conhecidas do Java, como `BigDecimal` e `BigInteger`. Vale lembrar que, para aplicar *fluent interface*, não necessariamente o objeto precisa ser imutável, como é o caso das classes `StringBuffer` e `StringBuilder`.

Criando uma classe imutável

Para que uma classe seja imutável, ela precisa atender a algumas características

[11]:

- Nenhum método pode modificar seu estado;
- A classe deve ser `final` para evitar que filhas permitam mutabilidade;
- Os atributos devem ser privados;
- Os atributos devem ser `final`, apenas para legibilidade de código, já que não há métodos que modificam o estado do objeto;
- Caso sua classe tenha composições com objetos mutáveis, eles devem ter acesso exclusivo pela sua classe (devolvendo cópias defensivas quando necessário).

Uma classe imutável ficaria como:

```
public final class Ano {
    private final int ano;
    public Ano(int ano) {
        this.ano = ano;
    }

    public int getAno() {
        return this.ano;
    }
}
```

O código começa a ficar mais complicado quando nossa classe imutável depende de objetos mutáveis. Por exemplo, uma classe `Periodo` imutável que, em Java, seja implementada através de `Calendar`s.

É preciso trabalhar com cópias defensivas em dois momentos. Primeiro quando recebemos algum `Calendar` como parâmetro e vamos armazená-lo internamente. E depois quando formos devolver algum `Calendar` que faça parte do estado do objeto. Se não fizermos as cópias, é possível quealgum código externo a nossa classe `Periodo` altere os `Calendar`s em questão.

Em código: (note as cópias defensivas no construtor e nos getters)

```
public final class Periodo {

    private final Calendar inicio;
    private final Calendar fim;

    public Periodo(Calendar inicio, Calendar fim) {
        this.inicio = (Calendar) inicio.clone();
        this.fim = (Calendar) fim.clone();
    }

    public Calendar getInicio() {
        return (Calendar) inicio.clone();
    }
```

```
    public Calendar getFim() {
        return (Calendar) fim.clone();
    }
}
```

Aproveitamos aqui que `Calendar` implementa `Cloneable`, caso contrário precisáriamos fazer a cópia manualmente: criando um novo objeto e alterando os atributos pertinentes.

Como nossa classe é imutável, se precisarmos de alguma "modificação", na verdade criamos um objeto novo. Podemos, por exemplo, criar um método que adie o período em questão em uma semana, ou seja, some sete dias ao fim do período:

```
public Periodo adiaUmaSemana() {
    Calendar novoFim = (Calendar) this.fim.clone();
    novoFim.add(Calendar.DAY_OF_MONTH, 7);
    return new Periodo(inicio, novoFim);
}
```

E, com uma pequena modificação, podemos implementar o design pattern *flyweight* em nossa classe e compartilhar a instância do `Calendar` de início do período entre o objeto original e o objeto novo com uma semana adiada. Para isso, precisaríamos de um outro construtor privado para ser chamado no `adiaUmaSemana` e que não faça o `clone`.

Como vimos, criar objetos imutáveis é muito simples. E com eles conseguimos uma série de benefícios, como simplicidade de código, thread-safety e possibilidades para melhor aproveitamento da memória. **Considere criar sua classe como imutável**.