correr o risco de cair na armadilha da quebra de encapsulamento e alto acoplamento. Repare que, desta forma, o acoplamento é bem menor, nenhuma das classes precisa conhecer o funcionamento interno das outras.

Porém, se for necessário o uso de herança, alguns cuidados são importantes para minimizar a possível quebra de encapsulamento. Joshua Bloch, no *Effective Java*, fala de *Design for Inheritance,*[4] com diversas práticas como evitar invocações entre métodos públicos, para que a sobrescrita de um não mude o comportamento de outro.

Ao mesmo tempo que desejamos evitar herança, temos interesse em permitir mudança no nosso comportamento sem a necessidade de alterar ou copiar o código original. Mantendo pontos de extensão, permitimos que classes que implementam uma determinada interface sejam capazes de modificar o comportamento de outras classes. Através de interfaces e composição, podemos criar desde um novo driver JDBC para um determinado banco de dados, ou flexibilizar trabalhos simples, como ordernar listas com `Comparators` implementando diferentes critérios de comparação. Este princípio, de manter suas classes abertas para extensão sem a necessidade de alteração no código original, é chamado *Open Closed Principle*.[13]

Evite herança, favoreça composição é um de dois princípios fundamentais de design do livro *Design Patterns*, com frequência referenciado na literatura.[6,3]

3.4. Favoreça imutabilidade e simplicidade

Dois clientes com a mesma data de pagamento não podem compartilhar instâncias de `Calendar`, pois, se um deles alterar o vencimento, essa mudança resultaria em um efeito colateral provavelmente indesejado: o outro também sofre a alteração. A classe `Calendar` permite mudança no estado de suas instâncias e, portanto, diz-se que a classe é mutável, como o exemplo a seguir demonstra:

```
Calendar data = new GregorianCalendar(2004, 2, 13);
cliente.setVencimento(data);

Calendar doisDiasDepois = cliente.getVencimento();
doisDiasDepois.add(Calendar.DAY_OF_MONTH, 2);

outroCliente.setVencimento(doisDiasDepois);

// efeito colateral:
System.out.println(cliente.getVencimento());
```

A classe `String` tem comportamento diferente. Quando começamos a programar com Java, passamos por um código semelhante ao que segue:

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

Este código, que aparece com frequência nas provas de certificação, imprimirá `java` em letras minúsculas. Isso porque a classe `String` é dita imutável. Todo método invocado em uma `String`, que parece modificá-la, sempre devolve uma nova instância com a alteração requisitada, mas nunca altera a instância original. Ao tornar uma classe imutável, uma ampla gama de problemas comuns desaparece.[14,4]

Simplicidade e previsibilidade

Objetos imutáveis são muito mais simples de manipular que os mutáveis, tendo um comportamento bem previsível:

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

Não importa o que o método `removeCaracteres(s)` faça, a saída apresentará *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();
verificaFeriado(c);
System.out.println(c.get(Calendar.YEAR));
```

Por mais explícito que seja o nome do método, não é possível afirmar, apenas com este trecho de código, qual é a saída no console, pois o método `verificaFeriado` pode alterar algum dado desta instância de `Calendar`. Os próprios engenheiros da Sun admitiram alguns erros de design das APIs antigas, como a classe `java.util.Calendar` ser mutável.[15] Por isso, muitas vezes recorremos às APIs de terceiros, como a **Joda Time**, nas quais encontramos entidades de datas, horários e intervalos imutáveis.

Quando o objeto é mutável, para evitar que alguém o modifique, temos que tomar alguma precaução. Uma solução, frequentemente usada com coleções através do

`Collections.unmodifiableList(List)` e seus similares, é interfacear o objeto e embrulhá-lo de tal forma que os métodos expostos não possibilitem modificação. Em vez de passar a referência original adiante, passamos essa nova referência, cuja instância lança exceções em qualquer tentativa de modificação.

Outra solução para objetos mutáveis é criar **cópias defensivas** do objeto. Em vez de devolver o objeto original que se deseja preservar, criamos uma cópia, por exemplo, através do método `clone`, e devolvemos esta cópia. O usuário, desta forma, não tem como alterar o objeto original, mas recebe uma cópia para uso próprio, que, se alterar, afetará apenas a si mesmo.

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. Objetos mutáveis, em contrapartida, com frequência necessitam de um cuidado adicional, cujo objetivo é evitar alterações inesperadas.

Otimizações de memória

Podemos tirar proveito da imutabilidade de outras formas.[14] Como cada objeto representa apenas um estado, você não precisa mais do que uma instância para cada estado. A própria API da linguagem Java usa a imutabilidade como uma vantagem para fazer cache e reaproveitamento de instâncias.

É o caso do *pool* de `Strings` da JVM, que faz com que `Strings` de mesmo conteúdo possam ser representadas por uma única instância compartilhada. O mesmo acontece em boa parte das implementações das classes wrapper como `Integer`. Ao invocar `Integer.valueOf(int)`, a referência a `Integer` devolvida pode ser fruto de um cache interno de objetos com números mais frequentemente solicitados.[16]

Esse tipo de otimização só é possível com objetos imutáveis. É seguro compartilhar instâncias de `Strings` ou `Integers` com diferentes partes do programa, sem o risco de que uma alteração no objeto traga efeitos colaterais indesejados em outras partes do sistema.

E há mais possibilidades ainda para otimizações graças à imutabilidade. A classe `String` ainda se aproveita dessa característica para compartilhar seu array de `char` interno entre `Strings` diferentes. Se olharmos seu código fonte, veremos que ela possui três atributos principais: um `char[] value` e dois inteiros, `count` e `offset`;[17] estes inteiros servem para indicar o início e o fim da `String` dentro do array de `char`.

Seu método `substring(int,int)` leva em conta a imutabilidade das `Strings` e os dois inteiros que controlam início e fim para reaproveitar o array

de `char`. Quando pedimos uma determinada *substring*, em vez de o Java criar um novo array de `char` com o pedaço em questão, ele devolve um novo objeto `String`, que internamente compartilha a referência para o mesmo array de `char` que a `String` original, e tem apenas os seus dois índices ajustados. Ou seja, criar uma substring em Java é muito leve, sendo apenas uma questão de ajustar dois números inteiros (Figura 3.3).

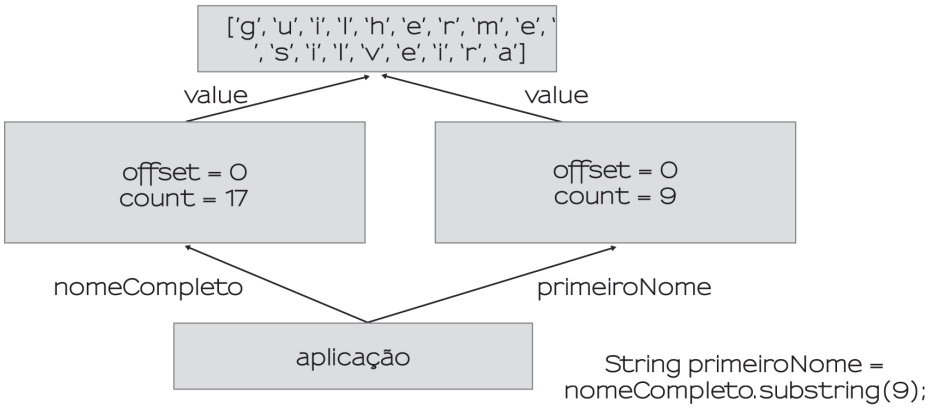

Figura 3.3 – Compartilhando o mesmo array entre duas Strings distintas.

Essa otimização é uma implementação um pouco mais simples do design pattern *flyweight*, em que se propõe reaproveitar objetos com objetivo de diminuir o uso da memória.[4] O próprio pool de Strings pode ser considerado um *flyweight*. E poderíamos ir mais longe com o *flyweight*, implementando a concatenação de Strings apontando para diversas outras Strings internamente ao invés de copiar os dados para seu próprio array de char. Neste caso, a API do Java prefere não chegar a este ponto, pois, apesar de um ganho de memória, haveria o *trade-off* do aumento de processamento.

É válido também ressaltar o perigo de certas otimizações em alguns casos. Na otimização do substring, por exemplo, uma `String` pequena pode acabar segurando referência para um array de `chars` muito grande se ela foi originada a partir de uma `String` longa. Isso impediria que o array maior fosse coletado, mesmo se não possuirmos referências para a `String` original.

Há também um excesso de memória consumida temporariamente. Com imutabilidade, cada invocação de método cria 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 pattern *builder* pode ajudar, como é o caso da classe mutável `StringBuilder` (ou sua versão

thread safe `StringBuffer`) com seu método `append` em vez de `String.concat` (ou o operador sobrecarregado +) em um laço.

Acesso por várias threads

Definir regiões críticas com `synchronized` é ainda uma solução repetidamente encontrada ao manipular memória compartilhada. Desta forma, evitamos que duas escritas concorrentes se entrelacem e que leituras sejam capazes de acessar dados inconsistentes ou intermediários.

A tarefa difícil é definir todas as regiões críticas e quais são mutuamente exclusivas; definir uma região muito grande gera perda de vazão (*throughtput*), enquanto uma de tamanho insuficiente implicará os mesmos problemas de não tê-las.

Em vez de recorrer aos recursos das linguagens, pensemos nesses objetos de forma diferente, evitando esse estado intermediário, não permitindo a mudança de valores. A vantagem é a *thread-safety*, pois, como não existem estados intermediários, não há como acessar nem modificar dados em momentos de inconsistência. O estado inconsistente fica escondido na lógica de construção, e o novo estado só estará disponível quando terminar de ser construído, para então ter sua referência compartilhada por mais de uma thread.

As linguagens funcionais mais puras, como *Erlang* e *Haskell*, estão sendo utilizadas 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. Muitas dessas linguagens podem rodar sobre a JVM, como Scala, que, apesar de suportar mutabilidade, tem fortes raízes em programação funcional e recomenda o uso de imutabilidade.

Imutabilidade e encadeamento de invocações de métodos

Uma vez que as classes são imutáveis, os métodos não possuem efeitos colaterais e, portanto, devem devolver uma referência a um novo objeto (Figura 3.4). Logo, é natural encadear invocações de métodos (*method chaining*), viabilizando a criação de código mais legível:

```
Data vencimento = new Data();
Calendar proximaParcela = vencimento.adicionaMes(1)
                                    .proximoDiaUtil()
                                    .toCalendar();
```

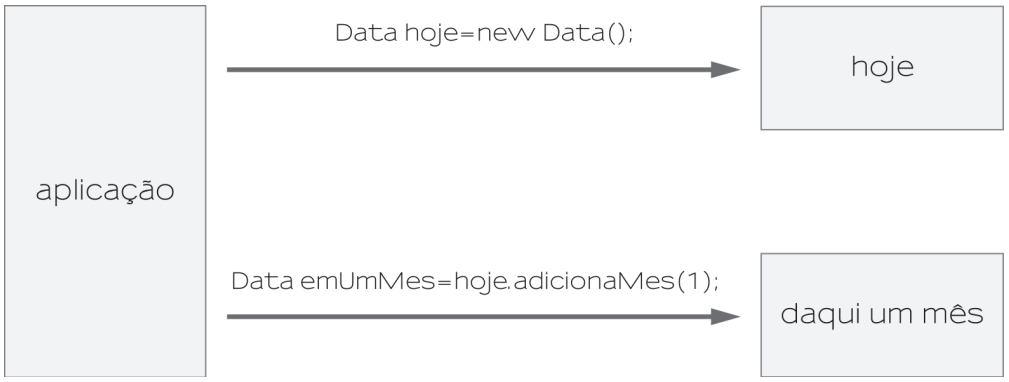

Figura 3.4 – Cada invocação de método que cria uma situação nova, cria um estado novo, devolve a referência para um objeto novo.

O mesmo acontece com outras classes imutáveis do Java, como `BigDecimal`, `BigInteger` e `String`:

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

Method chaining é uma prática simples que permite a criação de interfaces fluentes (*fluent interfaces*)[18] que, junto com outros padrões e aplicada a domínios específicos, permite a criação de *DSLs*.

Criando uma classe imutável

Para que uma classe seja imutável, ela precisa atender a algumas características:[4]

- Nenhum método pode modificar seu estado.
- Ela deve ser `final`, para evitar que filhas permitam mutabilidade.
- Os atributos devem ser privados.
- Os atributos devem ser `final`.
- 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;
    }
}
```

Este código seria mais complicado se nossa classe imutável fosse composta de objetos mutáveis. Uma classe `Periodo` imutável, implementada com `Calendars`, precisará trabalhar com cópias defensivas em dois momentos. Primeiro, quando receber algum `Calendar` como parâmetro e, depois, quando devolver algum `Calendar` que faça parte da composição do objeto. Se não copiarmos os objetos, é possível que algum código externo à classe `Periodo` altere os `Calendars` em questão, gerando inconsistência.

Note as cópias defensivas no construtor e nos getters no seguinte código:

```
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 o fato de `Calendar` implementar `Cloneable`, caso contrário precisaríamos fazer a cópia manualmente, criando um novo objeto e alterando os atributos pertinentes um a um.

Como nossa classe é imutável, se precisarmos calcular alguma informação que exija qualquer modificação, clonamos o objeto. É o caso de um método que adie o período 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, compartilhando a instância do `Calendar` de início do, período entre o objeto original e o novo, com uma semana adiada. Para tanto, precisaríamos de um outro construtor privado para ser chamado no `adiaUmaSemana` que não fizesse o `clone`.

Utilizar classes imutáveis traz um trabalho a mais junto com os diversos benefícios descritos. **Você deve considerar fortemente criar sua classe como imutável.**

3.5. Cuidado com o modelo anêmico

Um dos conceitos fundamentais da orientação a objetos é o de que você não deve expor seus detalhes de implementação. Encapsulando a implementação, podemos trocá-la com facilidade, já que não existe outro código dependendo desses detalhes, e o usuário só pode acessar seu objeto através do contrato definido pela sua interface pública.[19]

Costumeiramente, aprendemos que o primeiro passo nessa direção é declarar todos seus atributos como `private`:

```
public class Conta {
    private double limite;
    private double saldo;
}
```

Para acessar esses atributos, um desenvolvedor que ainda esteja aprendendo vai rapidamente cair em algum tutorial, que sugere a criação de *getters* e *setters* para poder trabalhar com esses atributos:

```
public class Conta {
    private double limite;
    private double saldo;

    public double getSaldo() {
        return saldo;
    }
```