Joshua Bloch

Java Efetivo

Terceira Edição

As Melhores Práticas para

... a Plataforma Java

ALTA BOOKS
EDITORA
Rio de Janeiro, 2019

Sumário

CAPÍTULO 1

Introdução

Este livro foi elaborado para ajudá-lo a utilizar de modo efetivo a linguagem e as bibliotecas de programação Java: `java.lang`, `java.util` e `java.io`, e os subpacotes Java, tais como, o `java.util.concurrent` e o `java.util.function`. Outras bibliotecas são discutidas eventualmente.

Esta obra é composta de 90 Itens, e cada um deles aborda o conhecimento de uma regra. Geralmente, as regras descrevem as práticas consideradas produtivas pelos melhores e mais experientes programadores. Os Itens são agrupados em 11 capítulos, cada qual abrange um aspecto amplo da arquitetura de software. A finalidade da obra não é ser lida do princípio ao fim: cada Item é mais ou menos independente. Os Itens estão amplamente correlacionados; desse modo, você pode facilmente trilhar o próprio caminho através das páginas.

Introduziram-se muitos recursos à plataforma desde a publicação da última edição deste livro. A maioria dos Itens desta obra usa de alguma forma essas funcionalidades. A tabela a seguir lhe mostra onde encontrar as principais funcionalidades elencadas:

Funcionalidade	**Itens**	**Versão**
Lambdas	Itens 42–44	Java 8
Streams	Itens 45–48	Java 8
Opcionais	Item 55	Java 8
Métodos padrões nas interfaces	Item 21	Java 8
try-with-resources	Item 9	Java 7
`@SafeVarargs`	Item 32	Java 7
Modules	Item 15	Java 9

A maioria dos Itens demonstra exemplos de programas. A característica fundamental deste livro reside nos exemplos de códigos que ilustram muitos padrões de projetos e práticas correntes de uso. Quando oportuno, eles são correlacionados às obras de referência padrão nessa área [Gamma95].

Muitos Itens apresentam um ou mais exemplos de programas que ilustram algumas práticas a serem evitadas. Esses exemplos, às vezes conhecidos como *antipadrões* [*antipatterns*], são identificados nitidamente com o comentário: `// Nunca faça isso!` Em cada caso, o Item explica por que o exemplo é errado e uma abordagem alternativa é sugerida.

Esta obra não se destina a iniciantes: pressupõe-se que você já esteja familiarizado com o Java. Caso não esteja, considere a leitura de um dos muitos textos introdutórios e excelentes, como o *Java Precisely*, de Peter Sestoft [Sestoft16]. Embora o *Java Efetivo* tenha sido idealizado para ser acessível a qualquer pessoa com um conhecimento prático da linguagem, também deve proporcionar estímulo à reflexão, mesmo para os programadores avançados.

A maior parte das regras deste livro se origina de alguns princípios fundamentais. A clareza e a simplicidade são de suma importância. O comportamento de um componente nunca deve surpreender o usuário, ele deve ser tão pequeno quanto possível, mas não muito pequeno. (Conforme empregado neste livro, o termo *componente* refere-se a qualquer elemento de programa reutilizável, partindo de um método individual a um framework complexo, constituído de vários pacotes.) O código deve ser reutilizado, em vez de copiado. As dependências entre os componentes devem ser as mínimas possíveis. Os erros devem ser detectados assim que possível e logo depois de serem cometidos, de preferência no momento da compilação.

Ainda que as regras da obra não se apliquem 100% das vezes, na maioria dos casos, descrevem as melhores práticas de programação. Você não as deve seguir cegamente; porém, deve violá-las apenas em certas ocasiões, e por um bom motivo. Aprender a arte da programação, como a maioria das outras disciplinas, consiste em primeiro aprender as regras e, depois, infringi-las.

De modo geral, este livro não trata de desempenho. Trata de escrever programas claros, adequados, utilizáveis, robustos, flexíveis e sustentáveis. Quando você consegue isso, obter o desempenho necessário costuma ser uma questão relativamente simples (Item 67). Alguns Itens não deixam de analisar os problemas de desempenho, e alguns desses Itens apresentam os cálculos de desempenho. Esses números, que são introduzidos pela frase “Na minha máquina”, devem ser considerados, na melhor das hipóteses, como valores aproximados.

Parece incrível, porém, a minha máquina é antiga, montada em casa com um processador de quatro núcleos Intel Core i7-4770K de 3,5 GHz, de 16 gigabytes de DDR3-1866 CL9 RAM, e executa a versão 9.0.0.15 do *Open Java SE Developement Kit (JDK)*, da Sun, no Microsoft Windows 7 Professional SP1 (64 bit).

Ao abordar as funcionalidades da linguagem de programação Java e suas bibliotecas, às vezes, é necessário consultar versões específicas. Por uma questão de conveniência, usamos codinomes em vez dos nomes das versões oficiais. Esta tabela mostra o mapeamento entre o nome das versões oficiais e os codinomes:

Nome da Versão Oficial	**Codinome**
JDK 1.0.x	Java 1.0
JDK 1.1.x	Java 1.1
Java 2 Platform, Standard Edition, v1.2	Java 2
Java 2 Platform, Standard Edition, v1.3	Java 3
Java 2 Platform, Standard Edition, v1.4	Java 4
Java 2 Platform, Standard Edition, v5.0	Java 5
Java Platform, Standard Edition 6	Java 6
Java Platform, Standard Edition 7	Java 7
Java Platform, Standard Edition 8	Java 8
Java Platform, Standard Edition 9	Java 9

Os exemplos são razoavelmente completos, todavia, privilegiam a legibilidade em vez da completude. Eles utilizam as classes dos pacotes `java.util` e `java.io` à vontade. A fim de compilar os exemplos, você pode ter que adicionar uma ou mais declarações de importação ou outro tipo de código boilerplate. O site do livro, em: `http://joshbloch.com/effectivejava`, apresenta uma versão expandida de cada exemplo, que você pode compilar e executar.

Na maioria dos casos, usamos os termos técnicos, conforme definidos no *The Java Language Specification, Java SE 8 Edition* [JLS]. Alguns termos merecem uma menção especial. A linguagem é compatível com quatro categorias de tipo: *interfaces* (incluindo as *anotações*), *classes* (incluindo as *enumerações*), *arrays* e *tipos primitivos.* As três primeiras são conhecidas como *tipos por referência*. As instâncias de classe e arrays são *objetos*, já os valores primitivos, não. *Os*

membros de uma classe são compostos por seus *campos*, *métodos*, *classes membro* e *interfaces membro*. A *assinatura* de um método é composta por seu nome e os tipos de seus parâmetros oficias; a assinatura *não* inclui o tipo de retorno do método.

Neste livro empregamos alguns termos de uma maneira diferente da qual aparecem na documentação *The Java Language Specification*. Ao contrário do *The Java Language Specification*, nosso livro usa *herança* como sinônimo de *derivação de subclasse*. Em vez de usarmos o termo herança para interfaces, simplesmente afirmamos que uma classe *implementa* uma interface ou que uma interface *estende* outra. O livro descreve o nível de acesso necessário quando não especificado usando o termo *pacote-privado* em vez do termo tecnicamente correto *acesso ao pacote* [JLS, 6.6.1].

O livro também emprega alguns termos técnicos não definidos no *The Java Language Specification*. O termo *API exportada*, ou simplesmente *API*, refere-se a classes, interfaces, construtores, membros e aos objetos serializados, através dos quais um programador acessa uma classe, interface ou pacote. (O termo *API*, abreviação de *application programming interface*, é mais utilizado do que o termo recomendado *interface*, para evitar a confusão relacionada ao conceito da linguagem suscitada por esse nome.) Um programador, ao escrever um programa, que por sua vez usa uma API, é chamado de *usuário* da API. Uma classe cuja implementação usa uma API é denominada como *cliente* da API.

Classes, interfaces, construtores, membros e objetos serializados são juntamente conhecidos como *elementos* da API. Uma API exportada é formada por elementos da API acessados fora do pacote que a define. Esses são os elementos da API que qualquer cliente pode usar, e para os quais o autor da API se compromete a dar suporte. Não por acaso, também são os elementos através dos quais o utilitário Javadoc gera a documentação a partir do modo de operação padrão. Falando de modo geral, a API exportada de um pacote é composta por membros públicos e protegidos e, construtores de todas as classes públicas ou interfaces do pacote.

No Java 9, adicionou-se um *sistema de módulos* à plataforma. Se uma biblioteca utilizar o sistema do módulo, sua API exportada é junção das APIs exportadas de todos os pacotes exportados pela declaração do módulo da biblioteca.

CAPÍTULO 2

Criar e Destruir Objetos

ESTE capítulo aborda a criação e a destruição de objetos: quando e como os criar, quando e como evitar os criar, como garantir que sejam destruídos no momento oportuno e como administrar quaisquer ações de limpeza que precedem a destruição dos objetos.

Item 1: Considere os métodos static factory em vez dos construtores

O modo tradicional para uma classe permitir que um cliente obtenha uma instância é fornecer um construtor público. Há outra técnica que deve fazer parte do conjunto do Java toolkit de todo programador. Uma classe pode fornecer um *método static factory* público, que é simplesmente um método estático que retorna uma instância de classe. A seguir, apresentamos um exemplo bem simples da classe `Boolean` (a classe que encapsula o tipo primitivo `boolean`). Esse método converte um valor do tipo primitivo `boolean` em um objeto de referência `Boolean`:

```
public static Boolean valueOf(boolean b)
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Observe que um método static factory não é o mesmo que o padrão *Método Factory* do *Design Patterns* [Gamma95]. O método static factory descrito nesse Item não tem equivalência direta no *Design Patterns*.

Uma classe pode fornecer a seus clientes métodos static factory em vez de, ou adicionalmente a, construtores públicos. Oferecer um método static factory em vez de um construtor público apresenta vantagens e desvantagens.

Uma das vantagens dos métodos static factory é que, ao contrário dos construtores, eles têm nomes. Se os parâmetros de um construtor não

descrevem categoricamente o objeto que está sendo retornado, é mais fácil usar um método static factory com um nome bem escolhido, e o resultante código do cliente também é mais fácil de ler. Por exemplo, o construtor `BigInteger(int, int, Random)`, que retorna um `BigInteger`, provavelmente primo, teria sido melhor projetado como um método static factory designado `BigInteger.probablePrime`. (Esse método foi introduzido no Java 4.)

Uma classe pode ter apenas um único construtor com uma determinada assinatura. Sabe-se que os programadores contornam essa restrição disponibilizando dois construtores cujas listas de parâmetros divergem apenas da ordem dos seus tipos de parâmetros. É uma péssima ideia. O usuário dessa API nunca se lembrará qual é o construtor certo e acabará chamando por engano o construtor errado. Ao lerem um código que usa esses construtores, as pessoas não saberiam o que fazem sem, antes, consultar a documentação da classe.

Como têm nomes, os métodos static factory não compartilham da restrição abordada no parágrafo anterior. Nos casos em que uma classe aparentemente exigir diversos construtores com a mesma assinatura, substitua os construtores por métodos static factory e por nomes escolhidos cuidadosamente, que ressaltem suas diferenças.

A segunda vantagem dos métodos static factory é que, ao contrário dos construtores, não precisam criar um novo objeto sempre que invocados. Isso permite que as classes imutáveis (Item 17) utilizem as instâncias pré-construídas ou armazenem em cache as instâncias, conforme são construídas, e as utilizem repetidas vezes a fim de evitar a criação de objetos duplicados desnecessários. O método `Boolean.valueOf(boolean)` exemplifica essa técnica: *nunca* cria um objeto. Essa técnica é semelhante ao padrão *Flyweigh* [Gamma95]. Ela melhora significativamente o desempenho caso os objetos equivalentes sejam requisitados com frequência, principalmente se a criação deles for custosa.

A capacidade dos métodos static factory de retornar o mesmo objeto a partir de chamadas repetidas possibilita às classes assegurarem o controle rigoroso sobre as instâncias existentes a todo momento. As classes que se comportam desse modo são denominadas de *classes controladoras de instância* (*instance-controlled*). Há várias razões para se escrever uma classe controladora de instâncias. O controle de instância permite que uma classe garanta que ela seja um singleton (Item 3) ou uma classe não instanciável (Item 4). Além disso, permite a uma classe de valor imutável (Item 17) assegurar que não existam duas instâncias iguais: `a.equals(b)` se e somente se `a == b`. Essa é a base do padrão *Flyweight* [Gamma95]. Os tipos enum (Item 34) proporcionam essa garantia.

A terceira vantagem dos métodos static factory é que, ao contrário dos construtores, podem retornar um objeto de qualquer subtipo do próprio tipo de retorno. Isso lhe dá uma grande flexibilidade na escolha da classe do objeto retornado.

Uma das aplicações dessa flexibilidade é que uma API consegue retornar objetos sem tornar suas classes públicas. Ao ocultar as classes de implementação, como resultado, temos uma API muito compacta. Essa técnica é compatível com os *frameworks baseados em interface* (Item 20), em que as interfaces são os tipos de retorno naturais para os métodos static factory.

Antes do Java 8, as interfaces não podiam ter métodos estáticos. Convencionalmente, os métodos static factory para uma interface denominada `Type` eram colocados em uma *classe complementar não instanciável* (Item 4) designada `Types`. Por exemplo, o Java Collections Framework apresenta 45 implementações utilitárias para suas interfaces, fornecendo coleções inalteráveis, sincronizadas e similares. Quase todas essas implementações são exportadas por intermédio de métodos static factory através de uma classe não instanciável (`java.util.Collections`). Nenhuma classe dos objetos retornados é pública.

A API do Collections Framework é bem menor do que teria sido caso exportasse 45 classes públicas separadas, uma para cada implementação pertinente. Não é apenas o *volume* da API que é reduzido, mas seu *peso conceitual*: a abundância e a dificuldade dos conceitos que os programadores devem dominar para usar a API. O programador sabe que o objeto retornado tem justamente a API especificada pela sua interface, dessa maneira, não há necessidade de se ler a documentação da classe adicional relacionada à classe de implementação. Ademais, o uso do método static factory exige que o cliente referencie o objeto retornado pela interface em vez de referenciar a classe de implementação, o que geralmente é uma boa prática (Item 64).

A partir do Java 8, aboliu-se a restrição de as interfaces não terem métodos estáticos, portanto, há poucas razões para fornecer uma classe complementar não instanciável a uma interface. Muitos membros estáticos públicos que costumam ser postos em tais classes deveriam ser colocados na própria interface. Observe, no entanto, que ainda pode ser necessário inserir a maior parte do código de implementação por trás desses métodos estáticos em uma classe pacote-privado separada. Isso ocorre porque o Java 8 exige que todos os membros estáticos de uma interface sejam públicos. O Java 9 permite a utilização de métodos estáticos privados, no entanto, exige-se que as classes membro estáticas e os campos estáticos ainda sejam públicos.

A quarta vantagem das static factories é que a classe do objeto retornado pode variar de chamada para chamada, em função dos parâmetros de entrada. Permite-se qualquer subtipo do tipo de retorno declarado. A classe do objeto retornado também pode variar de versão para versão.

A classe `EnumSet` (Item 36) não apresenta construtores públicos, somente static factories. Na implementação do OpenJDK, eles retornam uma instância de uma de duas subclasses, dependendo do tamanho do tipo enum subjacente: se tiver 64 elementos ou menos, como os tipos de enum têm, as static factories retornam uma instância `RegularEnumSet`, que são amparadas por um único `long`; caso o tipo enum tenha 65 elementos ou mais, as factories retornam uma instância `JumboEnumSet`, amparadas por um array `long`.

A existência dessas duas classes de implementação é invisível aos clientes. Se o `RegularEnumSet` deixasse de oferecer vantagens de desempenho aos tipos pequenos de enum, ele poderia até ser abolido em uma versão futura, sem quaisquer efeitos negativos. Do mesmo modo, uma versão futura poderia incorporar uma terceira ou quarta implementação do `EnumSet`, caso se comprovassem os benefícios para o desempenho. Os clientes não sabem e nem se preocupam com a classe do objeto que retornam da fábrica; eles só se preocupam que seja alguma subclasse do `EnumSet`.

A quinta vantagem das static factories é que não precisa existir a classe do objeto retornado quando a classe contém o método de escrita. Esses métodos static factory flexíveis formam a base do *service provider frameworks*, como a API do Java Database Connectivity (JDBC). Um service provider framework é um sistema no qual os provedores implementam um serviço, e o sistema disponibiliza as implementações aos clientes, desacoplando os clientes das implementações.

Há três componentes fundamentais em um service provider framework: *uma interface de serviço*, que representa uma implementação; *uma API de registro de provedores*, que os provedores usam para registrar as implementações; e uma *API de acesso ao serviço*, que os clientes utilizam para obter instâncias de serviço. A API de acesso ao serviço permite aos clientes especificarem os critérios ao escolher uma implementação. Na ausência desses critérios, a API retorna uma instância de implementação padrão, ou permite que o cliente alterne entre as implementações disponíveis. A API de acesso ao serviço é a static factory flexível, que constitui a base do service provider framework.

Um quarto componente opcional do service provider framework é uma interface de provedor de serviços que descreve um objeto de fabricação que produz as instâncias da interface de serviço. Na falta de uma interface de provedor de serviços, as implementações devem ser instanciadas reflexivamente

(Item 65). No caso do JDBC, o `Connection` desempenha o papel da interface de serviço, o `DriverManager.registerDriver` é a API de registro do provedor, o `DriverManager.getConnection` é a API de acesso ao serviço, e o `Driver` é a interface do provedor de serviços.

Há muitas variações do padrão do service provider framework. Por exemplo, a API de acesso ao serviço pode retornar uma interface de serviço mais caprichada aos clientes do que as APIs disponibilizadas pelos provedores. Esse é o padrão *Bridge* [Gamma95]. Os frameworks injetores de dependência (Item 5) podem ser considerados provedores de serviços poderosos. Desde o Java 6, a plataforma contempla um service provider framework para uso geral, o `java.util.ServiceLoader`, portanto, você não precisa, e geralmente não deve, escrever o próprio service provider framework (Item 59). O JDBC não usa o `ServiceLoader`, já que o primeiro é anterior ao segundo.

A principal limitação de fornecer apenas métodos static factory é que as classes sem construtores públicos ou protegidos não podem ser divididas em subclasses. Por exemplo, é impossível dividir em subclasses qualquer uma das classes pertinentes à implementação no Collections Framework. Provavelmente, isso pode ser uma dádiva disfarçada, pois, incentiva os programadores a usarem a composição em vez da herança (Item 18) exigida pelos tipos imutáveis (Item 17).

A segunda limitação dos métodos static factory é que são difíceis de ser encontrados pelos programadores. Eles não estão destacados na documentação da API do mesmo modo que os construtores, por esse motivo pode ser difícil descobrir como instanciar uma classe que forneça de métodos static factory em vez de construtores. Talvez, algum dia, a ferramenta Javadoc evidencie os métodos static factory. Enquanto isso, você minimiza esse problema ressaltando o uso das static factories na documentação na classe ou da interface, e respeitando as convenções comuns de nomenclatura. Aqui, estão alguns nomes comuns para os métodos static factory. É uma lista longe de ser exaustiva:

- **`from`** — Um *método de conversão de tipo* que apresenta um único parâmetro e retorna uma instância correspondente desse tipo, por exemplo:

```
Date d = Date.from(instant)
```

- **`of`** — Um *método de agregação* que apresenta diversos parâmetros e retorna uma instância desse tipo que incorpora esses parâmetros, por exemplo:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **`valueOf`** — Uma alternativa mais verbosa para o `from` e para o `of`, por exemplo:

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

- **`instance`** ou **`getInstance`** — Retorna uma instância que é descrita pelos seus parâmetros (se houver), mas não pode ter os mesmos valores, por exemplo:

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** ou **newInstance** — Análogo ao `instance` ou ao `getInstance`, embora, nesse caso, o método garanta que cada chamada retorne uma instância nova, por exemplo:

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **get*Type*** — Igual ao `getInstance`, porém é usado se o método de fabricação for de uma classe diferente. *Type* é o tipo de objeto retornado por um método de fabricação, por exemplo:

```
FileStore fs = Files.getFileStore(path);
```

- **new*Type*** — Igual ao `newInstance`, porém é usado se o método de fabricação for de uma classe diferente. *Type* é o tipo de objeto retornado por um método de fabricação, por exemplo:

```
BufferedReader br = Files.newBufferedReader(path)
```

- ***type*** — Uma alternativa concisa para o `get`*Type* e para o `new`*Type*, por exemplo:

```
List<Complaint> litany = Collections.list(legacyLitany)
```

Em suma, tanto os métodos static factory como os construtores públicos têm seus usos, e vale a pena compreender seus respectivos pontos positivos. Não raro, recomenda-se o uso dos métodos static factory, assim, evita-se o impulso de empregar construtores públicos sem antes levar em consideração as static factories.

Item 2: Cogite o uso de um builder quando se deparar com muitos parâmetros no construtor

As static factories e os construtores compartilham uma limitação: não se adéquam bem a um grande número de parâmetros opcionais. Analise o caso de uma classe, aqui exemplificada como `NutritionFacts`, representando um daqueles rótulos de informações nutricionais vinculados nas embalagens de comida. Eles têm alguns campos obrigatórios — a quantidade da porção, a quantidade das porções por embalagem e as calorias por porção — e mais de 20 campos opcionais — gorduras totais, gorduras saturadas, gorduras trans, colesterol, sódio e

assim por diante. A maioria dos produtos indica valores diferentes de zero em apenas um desses campos opcionais.

Quais tipos de construtores ou métodos static factory você deveria escrever para essa classe? Tradicionalmente, os programadores têm usado o padrão *telescoping constructor*, no qual você fornece um construtor somente com os parâmetros necessários, outro com um único parâmetro opcional, um terceiro com dois parâmetros opcionais, e assim por diante, resultando em um construtor com todos os parâmetros opcionais. Veja como isso funciona na prática. Para efeitos de síntese, apenas quatro campos opcionais são mostrados:

```
// Padrão telescoping construtor - não é escalável!
public class NutritionFacts {
    private final int servingSize;  // (mL)             exigido
    private final int servings;     // (per container) exigido
    private final int calories;     // (per serving)   opcional
    private final int fat;          // (g/serving)     opcional
    private final int sodium;       // (mg/serving)    opcional
    private final int carbohydrate; // (g/serving)     opcional

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings,
           int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

Quando quiser criar uma instância, você usa o construtor com a lista de parâmetros mais curta contendo todos os parâmetros que você queira definir:

```
NutritionFacts cocaCola =
    new NutritionFacts(240, 8, 100, 0, 35, 27);
```

Normalmente, essa invocação do construtor exigirá muitos parâmetros que você não quer atribuir, porém você é obrigado a definir um valor aos parâmetros de qualquer maneira. Nesse caso, atribuímos o valor de `0` para `fat`. Com "apenas" seis parâmetros, pode não ser tão ruim, mas isso sai fora de controle rapidamente à medida que o número de parâmetros aumenta.

Concluindo, **o padrão telescoping constructor funciona, mas é difícil escrever o código do cliente quando se tem muitos parâmetros, e é ainda mais difícil de o ler**. O leitor fica se perguntando o que significam todos esses valores, já que deve calcular cuidadosamente os parâmetros para descobrir. Sequências longas de parâmetros, quando digitadas de forma idêntica, podem causar bugs sutis. Se o cliente inverter acidentalmente dois desses parâmetros, o compilador não reclamará, mas o programa se comportará de modo errado em tempo de execução (Item 51).

Uma segunda alternativa para quando você se deparar com muitos parâmetros opcionais em um construtor é o padrão *JavaBeans*. Com ele, você chama um construtor sem parâmetros para criar o objeto e, em seguida, chama os métodos setter para definir cada parâmetro obrigatório e cada parâmetro opcional de interesse:

```
// Padrão JavaBeans - Permite a inconsistência, autoriza a mutabilidade
public class NutritionFacts {
    // Parâmetros inicializados para os valores padrões (se houver)
    private int servingSize  = -1; // Exigido; sem valor padrão
    private int servings     = -1; // Exigido; sem valor padrão
    private int calories     = 0;
    private int fat          = 0;
    private int sodium       = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }

    // Setters
    public void setServingSize(int val)  { servingSize = val; }
    public void setServings(int val)     { servings = val; }
    public void setCalories(int val)     { calories = val; }
    public void setFat(int val)          { fat = val; }
    public void setSodium(int val)       { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

Esse padrão não tem nenhuma das desvantagens que o padrão telescoping constructor apresenta. Com ele é fácil, embora um tanto prolixo, de criar instâncias, e de ler o código resultante:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Infelizmente, o padrão JavaBeans apresenta graves desvantagens. Como a construção é dividida em várias chamadas, **um JavaBean pode apresentar um estado parcialmente inconsistente durante sua construção.** A classe não tem a opção de implementar a consistência apenas verificando a validade dos parâmetros do construtor. As tentativas de usar um objeto quando ele apresenta um estado inconsistente podem causar falhas que estarão bem distantes do código com o bug e, consequentemente, serão difíceis de depurar. Uma das desvantagens relacionadas reside no fato de que **o padrão JavaBeans exclui a possibilidade de uma classe ser imutável** (Item 17) e exige, por parte do programador, um esforço complementar a fim de garantir a segurança da thread.

É possível minimizar essas desvantagens "congelando" manualmente o objeto quando sua construção está completa, e não permitindo o uso dele até estar congelado, porém essa variante é pesada e na prática é raramente usada. Inclusive, ela pode causar erros em tempo de execução, pois o compilador não garante que o programador chame do método de congelamento para um objeto antes de o usar.

Felizmente, há uma terceira alternativa, que combina a segurança do padrão telescoping constructor com a legibilidade do padrão JavaBeans. Trata-se do padrão *Builder* [Gamma95]. Em vez de construir diretamente o objeto pretendido, o cliente chama um construtor (ou uma static factory) com todos os parâmetros necessários e obtém um *objeto builder*. Em seguida, o cliente chama o método do tipo setter no objeto builder para definir cada parâmetro opcional de interesse. Por fim, o cliente chama um método `build` sem parâmetros para gerar o objeto, normalmente imutável. Em geral, o builder é um membro de classe estática (Item 24) da classe que ele constrói. Veja como isso funciona na prática:

```
// Padrão Builder
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Exige parâmetros
        private final int servingSize;
        private final int servings;

        // Parâmetros Opcionais - inicializado para os valores padrão
        private int calories      = 0;
        private int fat           = 0;
        private int sodium        = 0;
        private int carbohydrate  = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;           return this; }
        public Builder sodium(int val)
            { sodium = val;        return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val;  return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

A classe `NutritionFacts` é imutável, e todos os valores padrão dos parâmetros estão no mesmo lugar. Os métodos setter do builder retornam o próprio builder para as chamadas serem encadeadas, resultando em uma API *fluente*. Veja como é o código do cliente:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
        .calories(100).sodium(35).carbohydrate(27).build();
```

Esse código do cliente é fácil de escrever e, o mais importante, de ler. **O padrão Builder simula os parâmetros opcionais nomeados** como os que encontramos nas linguagens Python e Scala.

Por uma questão de síntese, omitiram-se as verificações de validação. Para identificar os parâmetros inválidos o mais rápido possível, verifique a validade dos parâmetros nos métodos do construtor e no builder. Confira as invariantes envolvendo parâmetros múltiplos no construtor chamado pelo método `build`. Para protegê-las contra ataques, realize as verificações nos campos do objeto, depois de copiar os parâmetros do builder (Item 50). Se a verificação falhar, lance uma `IllegalArgumentException` (Item 72), cuja mensagem detalhada indica quais parâmetros estão inválidos (Item 75).

O padrão Builder se adéqua bem às hierarquias de classe. Use uma hierarquia paralela de builders, cada qual aninhado à classe correspondente. As classes abstratas têm builders abstratos; as classes concretas, builders concretos. Por exemplo, considere uma classe abstrata na raiz de uma hierarquia, aqui representando vários tipos de pizza:

```
// Padrão Builder para hierarquias de classe
public abstract class Pizza {
   public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE
   final Set<Topping> toppings;

   abstract static class Builder<T extends Builder<T>> {
      EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
      public T addTopping(Topping topping) {
         toppings.add(Objects.requireNonNull(topping));
        return self();
      }

      abstract Pizza build();

      // As subclasses devem sobrescrever esse método para retornar "essa
      classe protegida abstrata T;self()
   }
   Pizza(Builder<?> builder) {
      toppings = builder.toppings.clone(); // Veja o Item 50
   }
}
```

Observe que `Pizza.Builder` é um *tipo genérico* com um *parâmetro do tipo recursivo* (Item 30). Isso, juntamente com o método abstrato `self`, permite que o encadeamento do método funcione apropriadamente nas subclasses, sem a necessidade de fazer o cast. Devido ao Java carecer de um tipo de self, esse paleativo é uma prática corrente, conhecida como *tipo de self simulado.*