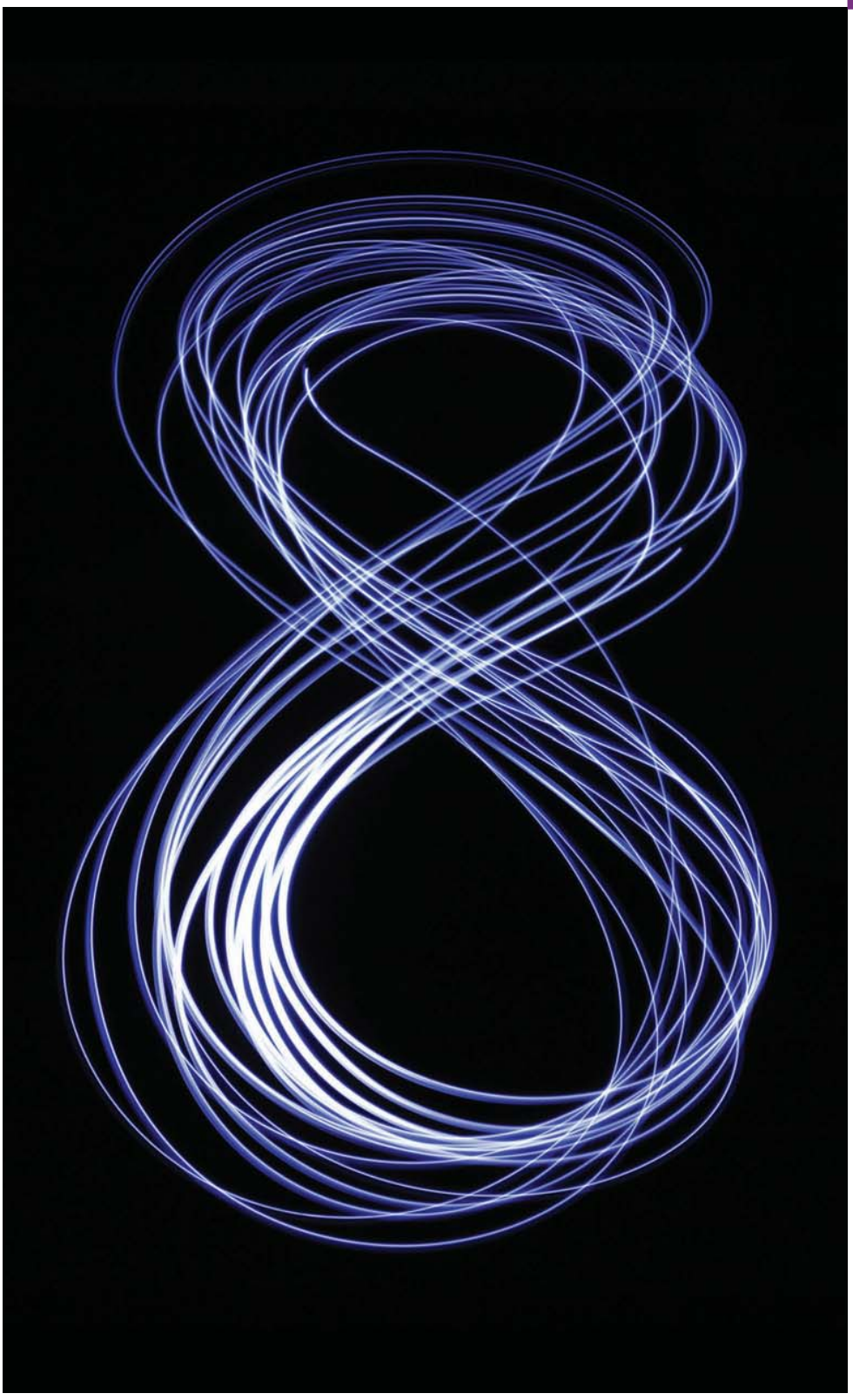
Introdução a classes, objetos, métodos e strings

3



Seus servidores públicos prestam-lhe bons serviços.
— Adlai E. Stevenson

Nada pode ter valor sem ser um objeto útil.
— Karl Marx

Objetivos

Neste capítulo, você irá:

- Descobrir como declarar uma classe e utilizá-la para criar um objeto.
- Ver como implementar comportamentos de uma classe como métodos.
- Aprender como implementar os atributos de uma classe como variáveis de instância.
- Verificar como chamar os métodos de um objeto para fazê-los realizarem suas tarefas.
- Detectar o que são variáveis locais de um método e como elas diferem de variáveis de instância.
- Distinguir o que são tipos primitivos e tipos de referência.
- Analisar como usar um construtor para inicializar dados de um objeto.
- Desvendar como representar e usar números contendo pontos decimais.

Sumário

3.1 Introdução

[*Observação*: este capítulo depende da terminologia e dos conceitos da programação orientada a objetos introduzida na Seção 1.5, "Introdução à tecnologia de objetos".]

No Capítulo 2, você trabalhou com classes, objetos e métodos *existentes*. Usou o objeto de saída padrão *predefinido* `System.out`, *invocando* seus métodos `print`, `println` e `printf` para exibir informações na tela. Empregou a classe `Scanner` *existente* a fim de criar um objeto que lê dados de números inteiros na memória inseridos pelo usuário com o teclado. Ao longo do livro, utilizará muito mais classes e objetos *preexistentes* — esse é um dos grandes pontos fortes do Java como uma linguagem de programação orientada a objetos.

Neste capítulo, você aprenderá a criar suas próprias classes e métodos. Cada nova *classe* que você desenvolve torna-se um novo *tipo* que pode ser utilizado para declarar variáveis e delinear objetos. Você pode declarar novos tipos de classe conforme necessário; essa é uma razão pela qual o Java é conhecido como uma linguagem *extensível*.

Apresentamos um estudo de caso sobre como criar e utilizar uma classe simples de uma conta bancária do mundo real, `Account`. Essa classe deve manter como *variáveis de instância* atributos como seu `name` e `balance` e fornecer *métodos* para tarefas como consulta de saldo (`getBalance`), fazer depósitos que aumentam o saldo (`deposit`) e realizar saques que diminuem o saldo (`withdraw`). Incorporaremos os métodos `getBalance` e `deposit` à classe nos exemplos do capítulo e você adicionará o método `withdraw` nos exercícios.

No Capítulo 2, utilizamos o tipo de dado `int` para representar números inteiros. Neste capítulo, introduziremos o tipo de dado `double` a fim de indicar um saldo de conta como um número que pode conter um *separador decimal* — esses números são chamados *números de ponto flutuante*. [No Capítulo 8, ao nos aprofundarmos um pouco mais na tecnologia de objetos, começaremos a representar valores monetários precisamente com a classe `BigDecimal` (pacote `java.math`), como você deve fazer ao escrever aplicativos monetários de força industrial.]

Normalmente, os aplicativos que você desenvolverá neste livro consistirão em duas ou mais classes. Se você se tornar parte de uma equipe de desenvolvimento na indústria, poderá trabalhar em aplicativos com centenas, ou até milhares, de classes.

3.2 Variáveis de instância, métodos *set* e métodos *get*

Nesta seção, você criará duas classes — `Account` (Figura 3.1) e `AccountTest` (Figura 3.2). A `AccountTest` é uma *classe de aplicativo* em que o método `main` criará e usará um objeto `Account` para demonstrar as capacidades da classe `Account`.

3.2.1 Classe `Account` com uma variável de instância, um método *set* e um método *get*

Diferentes contas normalmente têm diferentes nomes. Por essa razão, a classe `Account` (Figura 3.1) contém uma *variável de instância* `name`. Variáveis de instância de uma classe armazenam dados para cada objeto (isto é, cada instância) da classe. Mais adiante no capítulo adicionaremos uma variável de instância chamada `balance` a fim de monitorar quanto dinheiro está na conta. A classe `Account` contém dois métodos — o método `setName` armazena um nome em um objeto `Account` e o método `getName` obtém um nome de um objeto `Account`.

```
1  // Figura 3.1: Account.java
2  // Classe Account que contém uma variável de instância name
3  // e métodos para configurar e obter seu valor.
4
5  public class Account
6  {
7     private String name; // variável de instância
8
9     // método para definir o nome no objeto
10    public void setName(String name)
11    {
12       this.name = name; // armazena o nome
13    }
14
15    // método para recuperar o nome do objeto
16    public String getName()
17    {
18       return name; // retorna valor do nome para o chamador
19    }
20 } // fim da classe Account
```

Figura 3.1 | A classe `Account` que contém uma variável de instância `name` e métodos para *configurar* e *obter* seu valor.

Declaração de classe

A *declaração de classe* começa na linha 5:

```
public class Account
```

A palavra-chave `public` (explicada no Capítulo 8 em detalhes) é um **modificador de acesso**. Por enquanto, simplesmente declaramos toda classe `public`. Cada declaração de classe `public` deve ser armazenada em um arquivo com o *mesmo* nome que a classe e terminar com a extensão `.java`; do contrário, ocorrerá um erro de compilação. Assim, as classes `public` `Account` e `AccountTest` (Figura 3.2) *devem* ser declaradas nos arquivos *separados* `Account.java` e `AccountTest.java`, respectivamente.

Cada declaração de classe contém a palavra-chave `class` seguida imediatamente pelo nome da classe, nesse caso, `Account`. Cada corpo de classe é inserido entre um par de chaves esquerda e direita como nas linhas 6 e 20 da Figura 3.1.

Identificadores e nomeação usando a notação camelo

Nomes de classes, de método e de variável são *identificadores* e, por convenção, todos usam o mesmo esquema de nomeação com a *notação camelo* que discutimos no Capítulo 2. Também por convenção, os nomes de classe começam com uma letra *maiúscula*, e os de métodos e de variáveis iniciam com uma letra *minúscula*.

Variável de instância `name`

Lembre-se da Seção 1.5: um objeto tem atributos, implementados como variáveis de instância que o acompanham ao longo da sua vida. As variáveis de instância existem antes que os métodos sejam chamados em um objeto, enquanto eles são executados e depois que a execução deles foi concluída. Cada objeto (instância) da classe tem sua *própria* cópia das variáveis de instância da classe. Uma classe normalmente contém um ou mais métodos que manipulam as variáveis de instância pertencentes aos objetos particulares dela.

Variáveis de instância são declaradas *dentro* de uma declaração de classe, mas *fora* do corpo dos métodos da classe. A linha 7

```
private String name; // variável de instância
```

declara uma variável de instância `name` do tipo `String` *fora* do corpo dos métodos `setName` (linhas 10 a 13) e `getName` (linhas 16 a 19). Variáveis `String` podem conter valores de string de caracteres como `"Jane Green"`. Se houver muitos objetos `Account`, cada um tem seu próprio `name`. Como `name` é uma variável de instância, ele pode ser manipulado por cada um dos métodos da classe.

Boa prática de programação 3.1

Preferimos listar as variáveis de instância de uma classe primeiro no corpo dela, assim você pode ver o nome e o tipo das variáveis antes de elas serem utilizadas nos métodos da classe. Você pode listar as variáveis de instância da classe em qualquer lugar nela, fora das suas instruções de método, mas espalhar as variáveis de instância pode resultar em um código difícil de ler.

Modificadores de acesso `public` e `private`

A maioria das declarações de variável de instância é precedida pela palavra-chave `private` (como na linha 7). Da mesma forma que `public`, **`private`** é um *modificador de acesso.* As variáveis ou métodos declarados com o modificador de acesso `private` só são acessíveis a métodos da classe em que isso ocorre. Assim, a variável `name` só pode ser empregada nos métodos de cada objeto `Account` (nesse caso, `setName` e `getName`). Você verá mais adiante que isso apresenta oportunidades poderosas de engenharia de software.

Método `setName` da classe `Account`

Analisaremos o código da declaração do método `setName` (linhas 10 a 13):

```
public void setName(String name) – Esta linha é o cabeçalho do método
{
   this.name = name; // armazena o nome
}
```

Nós nos referimos à primeira linha de cada instrução de método (linha 10, nesse caso) como *cabeçalho do método.* O **tipo de retorno** do método (que aparece antes do nome deste) especifica a qualidade dos dados que o método retorna ao *chamador* depois de realizar sua tarefa. O tipo de retorno `void` (linha 10) indica que `setName` executará uma tarefa, mas *não* retornará (isto é, fornecerá) nenhuma informação ao seu chamador. No Capítulo 2, você usou métodos que retornam informações — por exemplo, adotou `Scanner` do método `nextInt` para inserir um número inteiro digitado pelo usuário no teclado. Quando `nextInt` lê um valor, ele o *retorna* para utilização no programa. Como veremos mais adiante, o método `Account getName` retorna um valor.

O método `setName` recebe um *parâmetro* `name` do tipo `String` — que representa o nome que será passado para o método como um *argumento.* Você verá como parâmetros e argumentos funcionam em conjunto ao discutir a chamada de método na linha 21 da Figura 3.2.

Os parâmetros são declarados em uma **lista de parâmetros** que está localizada entre os parênteses que seguem o nome do método no título dele. Quando existem múltiplos parâmetros, cada um é separado do seguinte por uma vírgula. Cada parâmetro *deve* especificar um tipo (nesse caso, `String`) seguido por um nome da variável (nesse caso, `name`).

Parâmetros são variáveis locais

No Capítulo 2, declaramos todas as variáveis de um aplicativo no método `main`. Variáveis declaradas no corpo de um método específico (como `main`) são **variáveis locais** que *somente* podem ser utilizadas nele. Cada método só pode acessar suas próprias variáveis locais, não aquelas dos outros. Quando esse método terminar, os valores de suas variáveis locais são *perdidos*. Os parâmetros de um método também são variáveis locais dele.

Corpo do método `setName`

Cada *corpo* de *método* é delimitado por um par de *chaves* (como nas linhas 11 e 13 da Figura 3.1) contendo uma ou mais instruções que executam tarefa(s) do método. Nesse caso, o corpo do método contém uma única instrução (linha 12) que atribui o valor do *parâmetro* `name` (uma `String`) à *variável de instância* `name` da classe, armazenando assim o nome da conta no objeto.

Se um método contiver uma variável local com o *mesmo* nome de uma variável de instância (como nas linhas 10 e 7, respectivamente), o corpo desse método irá referenciar a variável local em vez da variável de instância. Nesse caso, diz-se que a variável local *simula* a variável de instância no corpo do método. O corpo do método pode usar a palavra-chave **`this`** para referenciar a variável de instância simulada explicitamente, como mostrado à esquerda da atribuição na linha 12.

Boa prática de programação 3.2

Poderíamos ter evitado a necessidade da palavra-chave `this` aqui escolhendo um nome diferente para o parâmetro na linha 10, mas usar `this` como mostrado na linha 12 é uma prática amplamente aceita a fim de minimizar a proliferação de nomes de identificadores.

Após a linha 12 ser executada, o método completou sua tarefa, então ele retorna a seu *chamador.* Como você verá mais adiante, a instrução na linha 21 do `main` (Figura 3.2) chama o método `setName`.

Método `getName` da classe `Account`

O método `getName` (linhas 16 a 19)

```
public String getName()
{
   return name; // a palavra-chave return retorna o valor de name
                // para o método chamador
}
```

retorna ao chamador um `name` do objeto específico `Account`. O método tem uma lista *vazia* de parâmetros, então *não* exige informações adicionais para realizar sua tarefa. Ele retorna uma `String`. Quando um método que especifica um tipo de retorno *diferente* de `void` é chamado e conclui sua tarefa, ele *deve* retornar um resultado para seu chamador. Uma instrução que chama o método `getName` em um objeto `Account` (como aqueles nas linhas 16 e 26 da Figura 3.2) espera receber o nome de `Account` — uma `String`, como especificado no *tipo de retorno* da declaração do método.

A instrução `return` na linha 18 da Figura 3.1 passa o valor `String` da variável de instância `name` de volta ao chamador. Por exemplo, quando o valor é retornado para a instrução nas linhas 25 e 26 da Figura 3.2, a instrução o usa para gerar saída do nome.

3.2.2 Classe `AccountTest` que cria e usa um objeto da classe `Account`

Em seguida, gostaríamos de usar a classe `Account` em um aplicativo e *chamar* cada um dos seus métodos. Uma classe que contém um método `main` inicia a execução de um aplicativo Java. A classe `Account` *não pode* executar por si só porque *não* contém um método `main` — se digitar `java Account` na janela de comando, você obterá um erro indicando "`Main method not found in class Account`". Para corrigir esse problema, você deve declarar uma classe *separada* que contenha um método `main` ou colocar um método `main` na classe `Account`.

Classe `AccountTest` condutora

Para ajudá-lo a se preparar para os programas maiores que veremos futuramente neste livro e na indústria, usaremos uma classe `AccountTest` separada (Figura 3.2) contendo o método `main` a fim de testar a classe `Account`. Depois que começa a executar, `main` pode chamar outros métodos nessa e em outras classes; estas podem, por sua vez, chamar outros métodos etc. O método `main` da classe `AccountTest` cria um objeto `Account` e chama os métodos `getName` e `setName`. Essa classe é às vezes denominada *classe driver* (*ou "classe condutora"*) — assim como um objeto `Person` dirige um objeto `Car` informando-lhe o que fazer (ir mais rápido, ir mais devagar, virar à esquerda, virar à direita etc.), a classe `AccountTest` orienta um objeto `Account` indicando-lhe o que fazer ao chamar seus métodos.

```
1  // Figura 3.2: AccountTest.Java
2  // Cria e manipula um objeto Account.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // cria um objeto Scanner para obter entrada a partir da janela de comando
10       Scanner input = new Scanner(System.in);
11
12       // cria um objeto Account e o atribui a myAccount
13       Account myAccount = new Account();
14
15       // exibe o valor inicial do nome (null)
16       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
17
18       // solicita e lê o nome
19       System.out.println("Please enter the name:");
20       String theName = input.nextLine(); // lê uma linha de texto
21       myAccount.setName(theName); // insere theName em myAccount
22       System.out.println(); // gera saída de uma linha em branco
23
24       // exibe o nome armazenado no objeto myAccount
25       System.out.printf("Name in object myAccount is:%n%s%n",
26          myAccount.getName());
27    }
28 } // fim da classe AccountTest
```

```
Initial name is: null
Please enter the name:
Jane Green
Name in object myAccount is:
Jane Green
```

Figura 3.2 | Criando e manipulando um objeto `Account`.

Objeto Scanner para receber a entrada do usuário

A linha 10 cria um objeto `Scanner` chamado `input` para inserir o nome do usuário. Já a linha 19 exibe um prompt que pede para o usuário inserir um nome. E a linha 20 utiliza o método **`nextLine`** do objeto `Scanner` para ler o nome do usuário e atribuí-lo à variável *local* `theName`. Você digita o nome e pressiona *Enter* a fim de enviá-lo para o programa. Pressionar *Enter* insere um caractere de nova linha após aqueles digitados. O método `nextLine` lê os caracteres (incluindo o de espaço em branco, como em `"Jane Green"`) até encontrar a nova linha, então retorna uma `String` contendo os caracteres até, mas *não* incluindo, a nova linha, que é *descartada*.

A classe `Scanner` fornece vários outros métodos de entrada, como veremos ao longo do livro. Um método semelhante a `nextLine` — chamado **`next`** — lê a *próxima palavra*. Ao pressionar *Enter* depois de digitar algum texto, o método `next` lê os caracteres até encontrar um *caractere de espaço em branco* (como um espaço, tabulação ou nova linha), então retorna uma `String` contendo os caracteres até, mas *não* incluindo, o caractere de espaço em branco, que é *descartado*. Nenhuma informação depois do primeiro caractere de espaço em branco é *perdida* — elas podem ser lidas por outras instruções que chamam os métodos de `Scanner` posteriormente no programa.

Instância de um objeto — palavra-chave `new` e construtores

A linha 13 cria um objeto `Account` e o atribui à variável `myAccount` de tipo `Account`. A variável `myAccount` é inicializada com o resultado da **expressão de criação de instância de classe** `new Account()`. A palavra-chave **`new`** estabelece um novo objeto da classe especificada — nesse caso, `Account`. Os parênteses à direita de `Account` são *necessários*. Como veremos na Seção 3.4, esses parênteses em combinação com um nome de classe representam uma chamada para um **construtor**, que é *semelhante* a um método, mas é chamado implicitamente pelo operador `new` para *inicializar* as variáveis de instância de um objeto quando este é *criado*. Na Seção 3.4, você verá como colocar um *argumento* entre os parênteses para especificar um *valor inicial* a uma variável de instância `name` de um objeto `Account` — você aprimorará a classe `Account` para permitir isso. Por enquanto, simplesmente não incluímos *nada* entre os parênteses. A linha 10 contém uma expressão de criação de instância de classe para um objeto `Scanner` — a expressão inicializa o `Scanner` com `System.in`, que o informa de onde ler a entrada (isto é, o teclado).

Chamando o método `getName` da classe `Account`

A linha 16 exibe o nome *inicial*, que é obtido chamando o método `getName` do objeto. Assim como no caso do objeto `System.out` em relação a seus métodos `print`, `printf` e `println`, podemos também utilizar o objeto `myAccount` para chamar seus métodos `getName` e `setName`. A linha 16 chama `getName` usando o objeto `myAccount` criado na linha 13, seguido por um **ponto separador** (`.`), então o nome do método `getName` e um conjunto *vazio* de parênteses porque nenhum argumento está sendo passado. Quando `getName` é chamado

1. o aplicativo transfere a execução do programa a partir da chamada (linha 16 em `main`) para a declaração do método `getName` (linhas 16 a 19 da Figura 3.1). Como `getName` foi chamado via objeto `myAccount`, `getName` "sabe" qual variável de instância do objeto manipular.
2. então, o método `getName` executa sua tarefa, isto é, ele *retorna* o `name` (linha 18 da Figura 3.1). Quando a instrução `return` é executada, a execução do programa continua de onde `getName` foi chamado (linha 16 da Figura 3.2).
3. `System.out.printf` exibe a `String` retornada pelo método `getName`, então o programa continua executando na linha 19 em `main`.

Dica de prevenção de erro 3.1

Nunca use como controle de formato uma string inserida pelo usuário. Quando o método `System.out.printf` avalia a string de controle de formato no primeiro argumento, o método executa as tarefas com base no(s) especificador(es) de conversão nessa string. Se a string de controle de formato fosse obtida do usuário, alguém mal-intencionado poderia fornecer especificadores de conversão que seriam executados por `System.out.printf`, possivelmente causando uma falha de segurança.

`null` — o valor inicial padrão para variáveis `String`

A primeira linha da saída mostra o nome "`null`". Diferentemente das variáveis locais, que não *são* inicializadas de forma automática, *toda variável de instância tem um* **valor inicial padrão** — fornecido pelo Java quando você *não* especifica o valor inicial da variável de instância. Portanto, *não* é exigido que as *variáveis de instância* sejam explicitamente inicializadas antes de serem utilizadas em um programa — a menos que elas devam ser inicializadas para valores *diferentes* dos seus padrões. O valor padrão para uma variável de instância do tipo `String` (como `name` nesse exemplo) é **`null`**, que discutiremos mais adiante na Seção 3.3 ao abordar os *tipos por referência*.

Chamando o método `setName` da classe `Account`

A linha 21 chama o método `setName` de `myAccount`. Uma chamada de método pode fornecer *argumentos* cujos *valores* são atribuídos aos parâmetros de método correspondentes. Nesse caso, o valor da variável local de `main` entre parênteses é o *argumento* que é passado para `setName`, de modo que o método possa realizar sua tarefa. Quando `setName` é chamado:

1. o aplicativo transfere a execução do programa da linha 21 em `main` para a declaração do método `setName` (linhas 10 a 13 da Figura 3.1), e o *argumento valor* entre parênteses da chamada (`theName`) é atribuído ao *parâmetro* correspondente (`name`) no cabeçalho do método (linha 10 da Figura 3.1). Como `setName` foi chamado por objeto `myAccount`, `setName` "sabe" qual variável de instância do objeto manipular;
2. em seguida, o método `setName` executa sua tarefa — isto é, ele atribui o valor do parâmetro `name` à variável de instância `name` (linha 12 da Figura 3.1).
3. quando a execução do programa alcança a chave direita de fechamento de `setName`, ele retorna ao local onde `setName` foi chamado (linha 21 da Figura 3.2), então continua na linha 22 da Figura 3.2.

O número de *argumentos* na chamada de método *deve corresponder* ao de itens na lista de *parâmetros* da declaração do método. Além disso, os tipos de argumento na chamada de método precisam ser *consistentes* com os tipos de parâmetro correspondentes na declaração do método. (Como você aprenderá no Capítulo 6, nem sempre é requerido que um tipo de argumento e de seu parâmetro correspondente sejam *idênticos*.) No nosso exemplo, a chamada de método passa um argumento do tipo `String` (`theName`) — e a declaração do método especifica um parâmetro do tipo `String` (`name`, declarado na linha 10 da Figura 3.1). Portanto, nesse exemplo, o tipo de argumento na chamada de método equivale *exatamente* ao tipo de parâmetro no cabeçalho do método.

Exibindo o nome que foi inserido pelo usuário

A linha 22 da Figura 3.2 gera uma linha em branco. Quando a segunda chamada para o método `getName` (linha 26) é executada, o nome inserido pelo usuário na linha 20 é exibido. Já no momento em que a instrução nas linhas 25 e 26 conclui a execução, o final do método `main` é alcançado, assim, o programa termina.

3.2.3 Compilação e execução de um aplicativo com múltiplas classes

Você deve compilar as classes nas figuras 3.1 e 3.2 para que possa *executar* o aplicativo. Essa é a primeira vez que você criou um aplicativo com *múltiplas* classes. A classe `AccountTest` tem um método `main`, a classe `Account` não. Para compilar esse aplicativo, primeiro mude para o diretório que contém os arquivos do código-fonte dele. Em seguida, digite o comando

```
javac Account.java AccountTest.java
```

para compilar *ambas* as classes de uma vez. Se o diretório que contém o aplicativo incluir *apenas* os arquivos desse aplicativo, você pode compilar ambas as classes com o comando

```
javac *.java
```

O asterisco (*) em `*.java` indica que *todos* os arquivos no diretório *atual* que têm a extensão de nome de arquivo "`.java`" devem ser compilados. Se ambas as classes forem compiladas corretamente — isto é, nenhum erro de compilação for exibido — você pode então executar o aplicativo com o comando

```
java AccountTest
```

3.2.4 Diagrama de classe UML de `Account` com uma variável de instância e os métodos *set* e *get*

Utilizaremos com frequência os diagramas de classe UML para resumir os *atributos* e *operações* de uma classe. Na indústria, diagramas UML ajudam projetistas de sistemas a especificar um sistema de maneira gráfica, concisa e independente de linguagem de programação antes de os programadores implementarem o sistema em uma linguagem específica. A Figura 3.3 apresenta um **diagrama de classe UML** para a `Account` da Figura 3.1.

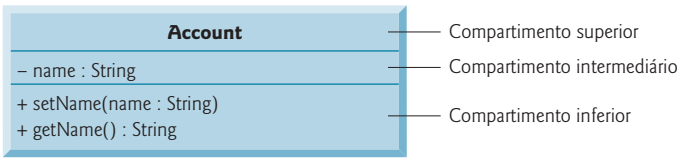


Figura 3.3 | Diagrama UML de classe para a `Account` da Figura 3.1.

Compartimento superior

Na UML, cada classe é modelada em um diagrama de classe como um retângulo com três compartimentos. Nesse diagrama, o compartimento *superior* contém o *nome da classe* `Account` centralizado horizontalmente em negrito.

Compartimento intermediário

O compartimento *intermediário* contém o *atributo* `name` *da classe*, que corresponde à variável de instância de mesmo nome em Java. A variável de instância `name` é `private` em Java, assim o diagrama UML de classe lista um *modificador de acesso com um sinal de subtração (–)* antes do nome do atributo. Depois do nome do atributo há um *dois pontos* e o *tipo de atributo,* nesse caso `String`.

Compartimento inferior

O compartimento *inferior* contém as **operações** da classe, `setName` e `getName`, que correspondem aos métodos com os mesmos nomes em Java. O UML modela as operações listando o nome de cada uma precedido por um *modificador de acesso,* nesse caso `+ getName`. Esse sinal de adição (+) indica que `getName` é uma operação *pública* na UML (porque é um método `public` em Java). A operação `getName` *não* tem nenhum parâmetro, então os parênteses após o nome dela no diagrama de classe estão *vazios,* assim como na declaração de método na linha 16 da Figura 3.1. A operação `setName`, também de caráter público, tem um parâmetro `String` chamado `name`.

Tipos de retorno

A UML indica o *tipo de retorno* de uma operação inserindo dois pontos e o tipo de retorno *após* os parênteses que vêm depois do nome da operação. O método `getName` de `Account` (Figura 3.1) tem um tipo de retorno `String`. O método `setName` *não* retorna um valor (porque retorna `void` em Java), então o diagrama de classe UML *não* especifica um tipo de retorno após os parênteses dessa operação.

Parâmetros

A UML modela um parâmetro de um modo pouco diferente do Java listando o nome desse parâmetro, seguido por dois-pontos e pelo tipo dele nos parênteses que seguem o nome da operação. O UML tem seus próprios tipos de dado semelhantes àqueles do Java, mas, para simplificar, usaremos os tipos de dado Java. O método `setName` de `Account` (Figura 3.1) tem um parâmetro `String` chamado `name`, assim a Figura 3.3 lista `name : String` entre parênteses após o nome do método.

3.2.5 Notas adicionais sobre a classe `AccountTest`

Método `static` `main`

No Capítulo 2, cada classe que declaramos tinha um método chamado `main`. Lembre-se de que `main` é um método especial que será *sempre* chamado automaticamente pela Java Virtual Machine (JVM) quando você executar um aplicativo. Você deve chamar a maioria dos outros métodos *explicitamente* para orientá-los a executar suas tarefas.

As linhas 7 a 27 da Figura 3.2 declaram o método `main`. Uma parte essencial para permitir à JVM localizar e chamar o método `main` a fim de iniciar a execução do aplicativo é a palavra-chave `static` (linha 7), que indica que `main` é um método `static`. O método `static` é especial, porque você pode chamá-lo *sem antes criar um objeto da classe na qual esse método é declarado* — nesse caso, a classe `AccountTest`. Discutiremos métodos `static` em detalhes no Capítulo 6.

Notas sobre declarações `import`

Note a declaração `import` na Figura 3.2 (linha 3), que indica ao compilador que o programa utiliza a classe `Scanner`. Como você aprendeu no Capítulo 2, as classes `System e String` estão no pacote `java.lang`, que é importado *implicitamente* para *todos* os programas Java, assim eles podem usar as classes desse pacote *sem* importá-las explicitamente. A *maioria* das outras classes que você empregará nos programas Java *precisa* ser importada *explicitamente*.

Há uma relação especial entre as classes que são compiladas no *mesmo* diretório, como as classes `Account` e `AccountTest`. Por padrão, essas classes são consideradas no *mesmo* pacote — conhecido como o **pacote padrão**. Classes no *mesmo* pacote são *importadas implicitamente* para os arquivos de código-fonte de outras classes nesse pacote. Assim, uma declaração `import` *não* é necessária quando uma classe adota outra no *mesmo* pacote — por exemplo, quando a classe `AccountTest` usa a classe `Account`.

A declaração `import` na linha 3 *não* é exigida se nos referimos à classe `Scanner` ao longo desse arquivo como `java.util.Scanner`, que inclui o *nome do pacote* e o *nome da classe completos.* Isso é conhecido como **nome de classe totalmente qualificado**. Por exemplo, a linha 10 da Figura 3.2 também pode ser escrita como

```
java.util.Scanner input = new java.util.Scanner(System.in);
```

Observação de engenharia de software 3.1

O compilador Java não requer declarações `import` *em um arquivo de código-fonte Java se o nome de classe totalmente qualificado for especificado sempre que um nome de classe é usado. A maioria dos programadores Java prefere o estilo de programação mais conciso que as declarações* `import` *fornecem.*

3.2.6 Engenharia de software com variáveis de instância `private` e métodos *set* e *get* `public`

Como veremos, usando os métodos *set e get,* você pode *validar* tentativas de modificações nos dados `private` e controlar como os dados são apresentados para o chamador — esses são benefícios convincentes da engenharia de software. Discutiremos isso com mais detalhes na Seção 3.5.

Se a variável de instância fosse `public`, qualquer **cliente** da classe — isto é, qualquer outra classe que chama os métodos de classe — poderia ver os dados e fazer o que quisesse com eles, inclusive configurá-los como um valor *inválido.*

Você talvez ache que, embora um cliente da classe não possa acessar diretamente uma variável de instância `private`, ele pode fazer o que quiser com a variável por meio dos métodos *set* e *get* `public`. Você talvez ache que é possível espiar os dados `private` a qualquer momento com o método *get* `public` e também modificá-los à vontade por meio do método *set* `public`. Mas os métodos *set* podem ser programados para *validar* seus argumentos e rejeitar qualquer tentativa de *definir* os dados como valores ruins, como temperatura corporal negativa, um dia em março fora do intervalo de 1 a 31, um código de produto que não está no catálogo da empresa etc. E um método *get* pode apresentar os dados de uma forma diferente. Por exemplo, uma classe `Grade` pode armazenar uma nota como um `int` entre 0 e 100, mas um método `getGrade` pode retornar uma classificação como uma `String`, por exemplo, `"A"` para as notas entre 90 e 100, `"B"` para as notas entre 80 e 89 etc. Controlar de perto o acesso e a apresentação dos dados `private` pode reduzir significativamente os erros, além de aumentar a robustez e a segurança dos seus programas.

A declaração de variáveis de instância com o modificador de acesso `private` é conhecida como *ocultamento de dados* ou *ocultamento de informações*. Quando um programa cria (instancia) um objeto de classe `Account`, a variável `name` é *encapsulada* (ocultada) no objeto e pode ser acessada apenas por métodos da classe do objeto.

Observação de engenharia de software 3.2

Anteceda cada variável de instância e declaração de método com um modificador de acesso. Geralmente, as variáveis de instância devem ser declaradas `private` *e os métodos,* `public`*. Mais adiante no livro, discutiremos por que você pode querer declarar um método* `private`*.*

Visualização conceitual de um objeto `Account` com dados encapsulados

Você pode pensar em um objeto `Account` como o mostrado na Figura 3.4. A variável de instância `private` chamada `name` permanece *oculta no* objeto (representado pelo círculo interno contendo `name`) e *protegida por uma camada externa* de métodos `public` (representados pelo círculo externo contendo `getName` e `setName`). Qualquer código do cliente que precisa interagir com o objeto `Account` *só* pode fazer isso chamando os métodos `public` da camada externa protetora.

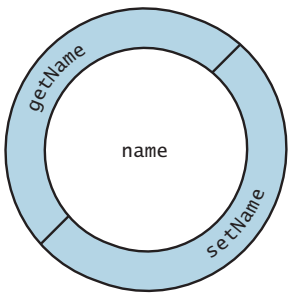


Figura 3.4 | Visualização conceitual de um objeto `Account` com sua variável de instância `private` chamada `name` encapsulada e com a camada protetora dos métodos `public`.

3.3 Tipos primitivos *versus* tipos por referência

Os tipos do Java são divididos em primitivos e **por referência**. No Capítulo 2, você trabalhou com variáveis do tipo `int` — um dos primitivos. Os outros tipos primitivos são `boolean`, `byte`, `char`, `short`, `long`, `float` e `double`, cada um dos quais discutiremos neste livro — eles estão resumidos no Apêndice D. Todos os tipos não primitivos são *por referência,* assim, as classes que especificam os objetos são por referência.

Uma variável de tipo primitivo pode armazenar exatamente *um* valor de seu tipo declarado por vez. Por exemplo, uma variável `int` pode armazenar um número inteiro de cada vez. Quando outro valor é atribuído a essa variável, ele substitui o anterior — que é *perdido.*

Lembre-se de que as variáveis locais *não* são inicializadas por padrão. Já as variáveis de instância de tipo primitivo *são* inicializadas por padrão — dos tipos `byte`, `char`, `short`, `int`, `long`, `float` e `double` como 0, e as do tipo `boolean` como `false`. Você pode especificar seu próprio valor inicial para uma variável do tipo primitivo atribuindo a ela um valor na sua declaração, como em

```
private int numberOfStudents = 10;
```

Os programas utilizam as variáveis de tipo por referência (normalmente chamadas **referências**) para armazenar as *localizações* de objetos na memória do computador. Dizemos que essa variável **referencia um objeto** no programa. *Objetos* que são referenciados podem conter *muitas* variáveis de instância. A linha 10 da Figura 3.2:

```
Scanner input = new Scanner(System.in);
```

cria um objeto da classe `Scanner`, então atribui à variável `input` uma *referência* a esse objeto `Scanner`. A linha 13 da Figura 3.2:

```
Account myAccount = new Account();
```

desenvolve um objeto da classe `Account`, então atribui à variável `myAccount` uma *referência* a esse objeto `Account`. *Variáveis de instância de tipo por referência, se não forem inicializadas explicitamente, o são por padrão para o valor* `null` — que representa uma "referência a nada". É por isso que a primeira chamada para `getName` na linha 16 da Figura 3.2 retorna `null` — o valor de `name` ainda *não* foi definido, assim o *valor padrão inicial* `null` é retornado.

Para chamar métodos em um objeto, você precisa de uma referência a ele. Na Figura 3.2, as instruções no método `main` usam a variável `myAccount` para chamar os métodos `getName` (linhas 16 e 26) e `setName` (linha 21) para interagir com o objeto `Account`. Variáveis de tipo primitivo *não* fazem referência a objetos, assim elas *não podem* ser usadas para chamar métodos.

3.4 Classe `Account`: inicialização de objetos com construtores

Como mencionado na Seção 3.2, quando um objeto de classe `Account` (Figura 3.1) é criado, sua variável de instância `String` chamada `name` é inicializada como `null` por *padrão*. Mas e se você quiser oferecer um nome ao *desenvolver* um objeto `Account`?

Cada classe que você declara tem como fornecer um *construtor* com parâmetros que podem ser utilizados para inicializar um objeto de uma classe quando o objeto for criado. O Java *requer* uma chamada de construtor para *cada* objeto que é desenvolvido, então esse é o ponto ideal para inicializar variáveis de instância de um objeto. O exemplo a seguir aprimora a classe `Account` (Figura 3.5) com um construtor que pode receber um nome e usá-lo para inicializar a variável de instância `name` quando um objeto `Account` é criado (Figura 3.6).

3.4.1 Declaração de um construtor `Account` para inicialização de objeto personalizado

Ao declarar uma classe, você pode fornecer seu próprio construtor a fim de especificar a *inicialização personalizada* para objetos de sua classe. Por exemplo, você pode querer especificar um nome para um objeto `Account` quando ele é criado, como na linha 10 da Figura 3.6:

```
Account account1 = new Account("Jane Green");
```

Nesse caso, o argumento `"Jane Green"` de `String` é passado para o construtor do objeto `Account` e é usado para inicializar a variável de instância `name`. A instrução anterior requer que a classe forneça um construtor que recebe apenas um parâmetro `String`. A Figura 3.5 contém uma classe `Account` modificada com esse construtor.

```
1  // Figura 3.5: Account.java
2  // a classe Account com um construtor que inicializa o nome.
3
4  public class Account
5  {
6     private String name; // variável de instância
7
8     // o construtor inicializa name com nome do parâmetro
9     public Account(String name) // o nome do construtor é nome da classe
10    {
```

continua

continuação

```
11          this.name = name;
12       }
13
14       // método para configurar o nome
15       public void setName(String name)
16       {
17          this.name = name;
18       }
19
20       // método para recuperar o nome do curso
21       public String getName()
22       {
23          return name;
24       }
25    } // fim da classe Account
```

Figura 3.5 | A classe `Account` com um construtor que inicializa o `name`.

Declaração do construtor de `Account`

As linhas 9 a 12 da Figura 3.5 declaram o construtor de `Account`. Um construtor *deve* ter o *mesmo nome* que a classe. Já uma *lista de parâmetros* de um construtor especifica que ele requer um ou mais dados para executar sua tarefa. A linha 9 indica que o construtor tem um parâmetro `String` chamado `name`. Ao criar um novo objeto `Account` (como veremos na Figura 3.6), você passará o nome de uma pessoa para o construtor, que receberá esse nome no parâmetro `name`. O construtor, então, atribuirá `name` à *instância variável* `name` na linha 11.

Dica de prevenção de erro 3.2

Embora seja possível fazer isso, não chame métodos a partir de construtores. Vamos explicar esse aspecto no Capítulo 10, Programação orientada a objetos: polimorfismo e interfaces.

Parâmetro `name` do construtor da classe `Account` e método `setName`

Lembre-se da Seção 3.2.1 que os parâmetros de método são variáveis locais. Na Figura 3.5, o construtor e o método `setName` têm um parâmetro chamado `name`. Embora esses parâmetros tenham o mesmo identificador (`name`), o parâmetro na linha 9 é uma variável local do construtor que *não* é visível para o método `setName`, e aquele na linha 15 é uma variável local de `setName` que *não* é visível para o construtor.

3.4.2 Classe `AccountTest`: inicialização de objetos `Account` quando eles são criados

O programa `AccountTest` (Figura 3.6) inicializa dois objetos `Account` usando o construtor. A linha 10 cria e inicializa o objeto `Account` denominado `account1`. A palavra-chave `new` solicita memória do sistema para armazenar o objeto `Account`, então chama implicitamente o construtor da classe correspondente para *inicializá-lo*. A chamada é indicada pelos parênteses após o nome da classe, que contêm o *argumento* `"Jane Green"` usado para inicializar o nome do novo objeto. A expressão de criação da instância de classe na linha 10 retorna uma *referência* ao novo objeto, que é atribuído à variável `account1`. A linha 11 repete esse processo, passando o argumento `"John Blue"` a fim de inicializar o nome para `account2`. As linhas 14 e 15 utilizam o método `getName` de cada objeto para obter os nomes e mostrar que eles, de fato, foram inicializados quando os objetos foram *criados*. A saída mostra nomes *diferentes,* confirmando que cada `Account` mantém sua *própria cópia* da variável de instância `name`.

```
1  // Figura 3.6: AccountTest.Java
2  // Usando o construtor de Account para inicializar a instância name
3  // variável no momento em que cada objeto Account é criado.
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // cria dois objetos Account
10       Account account1 = new Account("Jane Green");
11       Account account2 = new Account("John Blue");
12
13       // exibe o valor inicial de nome para cada Account
14       System.out.printf("account1 name is: %s%n", account1.getName());
```

continua

```
15          System.out.printf("account2 name is: %s%n", account2.getName());
16       }
17    } // fim da classe AccountTest
```

continuação

```
account1 name is: Jane Green
account2 name is: John Blue
```

Figura 3.6 | Usando o construtor de `Account` para inicializar a variável de instância `name` no momento em que cada objeto `Account` é criado.

Construtores não podem retornar valores

Uma diferença importante entre construtores e métodos é que os *construtores não podem retornar valores*, portanto, *não podem* especificar um tipo de retorno (nem mesmo `void`). Normalmente, os construtores são declarados `public` — mais adiante no livro explicaremos quando usar construtores `private`.

Construtor padrão

Lembre-se de que a linha 13 da Figura 3.2

```
Account myAccount = new Account();
```

usou `new` para criar um objeto `Account`. Os parênteses *vazios* depois de "`new Account`" indicam uma chamada para o **construtor padrão** da classe — em qualquer classe que *não* declare explicitamente um construtor, o compilador fornece um tipo padrão (que sempre não tem parâmetros). Quando uma classe tem somente o construtor padrão, as variáveis de instância da classe são inicializadas de acordo com seus *valores padrões*. Na Seção 8.5, você aprenderá que as classes podem ter múltiplos construtores.

Não há nenhum construtor padrão em uma classe que declara um construtor

Se você declarar um construtor para uma classe, o compilador *não* criará um *construtor padrão* para ela. Nesse caso, você não será capaz de estabelecer um objeto `Account` com a expressão de criação de instância da classe `new Account()`, como fizemos na Figura 3.2 — a menos que o construtor personalizado que você declare *não* receba nenhum parâmetro.

Observação de engenharia de software 3.3

A menos que a inicialização padrão de variáveis de instância de sua classe seja aceitável, forneça um construtor personalizado para assegurar que elas sejam adequadamente inicializadas com valores significativos quando cada novo objeto de sua classe for criado.

Adicionando o construtor ao diagrama de classe UML da classe `Account`

O diagrama de classe UML da Figura 3.7 modela a classe `Account` da Figura 3.5, que tem um construtor com um parâmetro `name` de `String`. Assim como as operações, a UML modela construtores no *terceiro* compartimento de um diagrama de classe. Para distinguir entre um construtor e as operações de uma classe, a UML requer que a palavra "constructor" seja colocada entre **aspas francesas (« e »)** antes do nome do construtor. É habitual listar construtores *antes* de outras operações no terceiro compartimento.

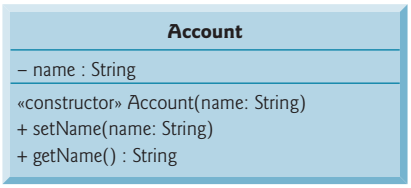


Figura 3.7 | Diagrama de classe UML para a classe `Account` da Figura 3.5.

3.5 A classe `Account` com um saldo; números de ponto flutuante

Agora declaramos uma classe `Account` que mantém o *saldo* de uma conta bancária além do nome. A maioria dos saldos das contas não é de números inteiros. Assim, a classe `Account` representa o saldo da conta como um **número de ponto flutuante** — com um *ponto decimal,* como 43,95, 0,0, –129,8873. [No Capítulo 8, começaremos representando valores monetários precisamente com a classe `BigDecimal`, como você deve fazer ao escrever aplicativos monetários de força industrial.]

O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória — `float` e `double`. As variáveis do tipo `float` representam **números de ponto flutuante de precisão simples** e podem ter até *sete dígitos significativos*. Já as variáveis de tipo `double` representam **números de ponto flutuante de dupla precisão**. Esses exigem *o dobro* de memória que variáveis `float` e podem conter até *15 dígitos significativos* — quase o *dobro* da precisão das variáveis `float`.

A maioria dos programadores representa números de ponto flutuante com o tipo `double`. De fato, o Java trata todos os números de ponto flutuante que você digita no código-fonte de um programa (7,33 e 0,0975, por exemplo) como valores `double` por padrão. Esses valores no código-fonte são conhecidos como **literais de ponto flutuante**. Veja o Apêndice D, Tipos primitivos, para informações sobre os intervalos precisos de valores de `float`s e `double`s.

3.5.1 A classe `Account` com uma variável de instância `balance` do tipo `double`

Nosso próximo aplicativo contém uma versão da classe `Account` (Figura 3.8) que mantém como variáveis de instância o `name` *e* o `balance` de uma conta bancária. Um banco típico atende *muitas* contas, cada uma com um saldo *próprio*, portanto, a linha 8 declara uma variável de instância chamada `balance` do tipo `double`. Cada instância (isto é, objeto) da classe `Account` contém suas *próprias* cópias *tanto* de `name` *como de* `balance`.

```
1  // Figura 3.8: Account.java
2  // Classe Account com uma variável de instância balance do tipo double e um construtor
3  // e método deposit que executa a validação.
4
5  public class Account
6  {
7     private String name; // variável de instância
8     private double balance; // variável de instância
9
10    // Construtor de Account que recebe dois parâmetros
11    public Account(String name, double balance)
12    {
13       this.name = name; // atribui name à variável de instância name
14
15       // valida que o balance é maior que 0.0; se não for,
16       // a variável de instância balance mantém seu valor inicial padrão de 0.0
17       if (balance > 0.0) // se o saldo for válido
18          this.balance = balance; // o atribui à variável de instância balance
19    }
20
21    // método que deposita (adiciona) apenas uma quantia válida no saldo
22    public void deposit(double depositAmount)
23    {
24       if (depositAmount > 0.0) // se depositAmount for válido
25          balance = balance + depositAmount; // o adiciona ao saldo
26    }
27
28    // método retorna o saldo da conta
29    public double getBalance()
30    {
31       return balance;
32    }
33
34    // método que define o nome
35    public void setName(String name)
36    {
37       this.name = name;
38    }
39
40    // método que retorna o nome
41    public String getName()
42    {
43       return name; // retorna o valor de name ao chamador
44    } // fim do método getName
45 } // fim da classe Account
```

Figura 3.8 | A classe `Account` com uma variável de instância `balance` do tipo `double`, um construtor e o método `deposit` que executa a validação.

Construtor com dois parâmetros da classe `Account`

A classe tem um *construtor* e quatro *métodos*. É comum que alguém que abre uma conta deposite o dinheiro imediatamente, assim o construtor (linhas 11 a 19) recebe um segundo parâmetro — `initialBalance` do tipo `double` que representa o *saldo inicial*. As linhas 17 e 18 asseguram que `initialBalance` seja maior do que `0.0`. Se for, o valor de `initialBalance` é atribuído à variável de instância `balance`. Caso contrário, `balance` permanece em `0.0` — seu *valor inicial padrão*.

Método `deposit` da classe `Account`

O método `deposit` (linhas 22 a 26) *não* retorna quaisquer dados quando ele completa sua tarefa, portanto, seu tipo de retorno é `void`. O método recebe um parâmetro nomeado `depositAmount` — um valor `double` que é *adicionado* a `balance` *apenas* se o parâmetro for *válido* (isto é, maior que zero). Primeiro, a linha 25 adiciona o `balance` atual e `depositAmount`, formando uma soma *temporária* que é *então* atribuída a `balance` *substituindo* seu valor anterior (lembre-se de que a adição tem precedência *maior* do que a atribuição). É importante entender que o cálculo à direita do operador de atribuição na linha 25 *não* modifica o saldo — por isso a atribuição é necessária.

Método `getBalance` da classe `Account`

O método `getBalance` (linhas 29 a 32) permite aos *clientes* da classe (isto é, outras classes cujos métodos chamam os dessa referida) obter o valor do `balance` de um objeto `Account` particular. O método especifica o tipo de retorno `double` e uma lista *vazia* de parâmetros.

Todos os métodos de `Account` podem utilizar `balance`

Mais uma vez, as instruções nas linhas 18, 25 e 31 empregam a variável `balance`, embora ela *não* tenha sido declarada em *nenhum* dos métodos. Podemos utilizar `balance` nesses métodos porque ele é uma *variável de instância* da classe.

3.5.2 A classe `AccountTest` para utilizar a classe `Account`

A classe `AccountTest` (Figura 3.9) cria dois objetos `Account` (linhas 9 e 10) e os inicializa com um saldo *válido* de `50.00` e um *inválido* de `-7.53`, respectivamente — para o propósito dos nossos exemplos, supomos que os saldos devem ser maiores ou iguais a zero. As chamadas para o método `System.out.printf` nas linhas 13 a 16 geram os nomes e os saldos das contas, que são obtidos chamando os métodos `getName` e `getBalance` de `Account`.

```
1  // Figura 3.9: AccountTest.Java
2  // Entrada e saída de números de ponto flutuante com objetos Account.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        Account account1 = new Account("Jane Green", 50.00);
10       Account account2 = new Account("John Blue", -7.53);
11
12       // exibe saldo inicial de cada objeto
13       System.out.printf("%s balance: $%.2f %n",
14          account1.getName(),account1.getBalance());
15       System.out.printf("%s balance: $%.2f %n%n",
16          account2.getName(),account2.getBalance());
17
18       // cria um Scanner para obter entrada a partir da janela de comando
19       Scanner input = new Scanner(System.in);
20
21       System.out.print("Enter deposit amount for account1: "); // prompt
22       double depositAmount = input.nextDouble(); // obtém a entrada do usuário
23       System.out.printf("%nadding %.2f to account1 balance%n%n",
24          depositAmount);
25       account1.deposit(depositAmount); // adiciona o saldo de account1
26
27       // exibe os saldos
28       System.out.printf("%s balance: $%.2f %n",
29          account1.getName(),account1.getBalance());
30       System.out.printf("%s balance: $%.2f %n%n",
31          account2.getName(),account2.getBalance());
32
```

continua

continuação

```
33          System.out.print("Enter deposit amount for account2: "); // prompt
34          depositAmount = input.nextDouble(); // obtém a entrada do usuário
35          System.out.printf("%nadding %.2f to account2 balance%n%n",
36             depositAmount);
37          account2.deposit(depositAmount); // adiciona ao saldo de account2
38
39          // exibe os saldos
40          System.out.printf("%s balance: $%.2f %n",
41             account1.getName(),account1.getBalance());
42          System.out.printf("%s balance: $%.2f %n%n",
43             account2.getName(),account2.getBalance());
44       } // fim de main
45    } // fim da classe AccountTest
```

```
Jane Green balance: $50.00
John Blue balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Jane Green balance: $75.53
John Blue balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Jane Green balance: $75.53
John Blue balance: $123.45
```

Figura 3.9 | Entrada e saída de números de ponto flutuante com objetos `Account`.

Exibição dos saldos iniciais dos objetos `Account`

Quando o método `getBalance` é chamado por `account1` a partir da linha 14, o valor do `balance` da `account1` é retornado da linha 31 da Figura 3.8 e exibido pela instrução `System.out.printf` (Figura 3.9, linhas 13 e 14). De maneira semelhante, quando o método `getBalance` for chamado por `account2` da linha 16, o valor do `balance` da `account2` é retornado da linha 31 da Figura 3.8 e exibido pela instrução `System.out.printf` (Figura 3.9, linhas 15 e 16). O `balance` de `account2` é inicialmente `0.00` porque o construtor rejeitou a tentativa de iniciar `account2` com um saldo *negativo,* assim o saldo retém seu valor inicial padrão.

Formatando números de ponto flutuante para exibição

Cada um dos `balances` é gerado por `printf` com o especificador de formato `%.2f`. O **especificador de formato `%f`** é utilizado para gerar saída de valores de tipo `float` ou `double`. O `.2` entre `%` e `f` representa o número de *casas decimais* (2) que devem ser colocadas à *direita* do ponto decimal no número de ponto flutuante — também conhecido como a **precisão** do número. Qualquer valor de ponto flutuante com `%.2f` será *arredondado* para a *casa decimal dos centésimos* — por exemplo, 123,457 se tornaria 123,46 e 27,33379 seria arredondado para 27,33.

Lendo um valor de ponto flutuante pelo usuário e realizando um depósito

A linha 21 (Figura 3.9) solicita que o usuário insira um valor de depósito para `account1`. Já a linha 22 declara a variável `depositAmount` *local* para armazenar cada montante de depósito inserido pelo usuário. Ao contrário das variáveis de *instância* (como `name` e `balance` na classe `Account`), variáveis *locais* (como `depositAmount` em `main`) *não* são inicializadas por padrão, então elas normalmente devem ser inicializadas de forma explícita. Como veremos mais adiante, o valor inicial da variável `depositAmount` será determinado pela entrada do usuário.

Erro comum de programação 3.1

O compilador Java emitirá um erro de compilação se você tentar usar o valor de uma variável local não inicializada. Isso ajuda a evitar erros perigosos de lógica no tempo de execução. Sempre é melhor remover os erros dos seus programas no tempo de compilação em vez de no tempo de execução.

A linha 22 obtém a entrada do usuário chamando o método **`nextDouble`** do objeto `Scanner input`, que retorna um valor `double` inserido pelo usuário. As linhas 23 e 24 exibem o `depositAmount`. Já a linha 25 chama o método `deposit` do objeto `account1` com o `depositAmount` como *argumento* desse método. Quando o método é chamado, o valor do argumento é atribuí-

do ao parâmetro `depositAmount` do método `deposit` (linha 22 da Figura 3.8); então o método `deposit` adiciona esse valor a `balance`. As linhas 28 a 31 (Figura 3.9) geram `name` e `balance` de ambas as `Account` *novamente* para mostrar que *só* `balance` da `account1` mudou.

A linha 33 pede ao usuário para inserir um valor de depósito para `account2`. Então, a linha 34 obtém a entrada do usuário chamando o método `nextDouble` do objeto `Scanner input`. As linhas 35 e 36 exibem o `depositAmount`. Ainda, a linha 37 chama o método `deposit` do objeto `account2` com o `depositAmount` como o *argumento* desse método; em seguida, o método `deposit` adiciona esse valor ao `balance`. Por fim, as linhas 40 a 43 geram `name` e `balance` de ambas as `Account` *novamente* para mostrar que *só* `account2` mudou.

Código duplicado no método `main`

As seis instruções nas linhas 13 e 14, 15 e 16, 28 e 29, 30 e 31, 40 e 41 e 42 e 43 são quase *idênticas* — cada uma delas gera um `name` e um `balance` da `Account`. Elas só diferem no nome do objeto `Account` — `account1` ou `account2`. Código duplicado como esse pode criar *problemas de manutenção de código* quando ele precisa ser atualizado — se todas as *seis* cópias do mesmo código tiverem o mesmo erro ou atualização a ser feita, você deve fazer essa mudança *seis* vezes, *sem cometer erros.* O Exercício 3.15 solicita que você modifique a Figura 3.9 para incluir um método `displayAccount` que recebe como parâmetro um objeto `Account` e gera `name` e `balance` dele. Você, então, substituirá as instruções duplicadas de `main` por seis chamadas para `displayAccount`, reduzindo, assim, o tamanho do seu programa e melhorando sua manutenciabilidade usando *uma* cópia do código que exibe um `name` e um `balance` de `Account`.

!

Observação de engenharia de software 3.4

Substituir o código duplicado por chamadas para um método que contém uma cópia dele pode reduzir o tamanho do seu programa e melhorar sua manutenciabilidade.

Diagrama de classe UML para a classe `Account`

O diagrama de classe UML na Figura 3.10 modela de maneira concisa a classe `Account` da Figura 3.8. Isso acontece no *segundo* compartimento com os atributos `private` `name` tipo `String` e `balance` do tipo `double`.

O *construtor* da classe `Account` é modelado no *terceiro* compartimento com os parâmetros `name` do tipo `String` e `initialBalance` do tipo `double`. Quatro métodos `public` da classe também são modelados no *terceiro* compartimento — a operação `deposit` com um parâmetro `depositAmount` do tipo `double`, a operação `getBalance` com um tipo de retorno `double`, a operação `setName` com um parâmetro `name` do tipo `String` e a operação `getName` com um tipo de retorno `String`.

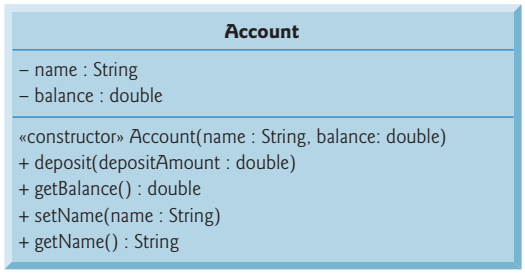


Figura 3.10 | Diagrama de classe UML para a classe `Account` da Figura 3.8.

3.6 (Opcional) Estudo de caso de GUIs e imagens gráficas: utilizando caixas de diálogo

Este estudo de caso opcional é projetado para aqueles que querem começar a aprender as poderosas capacidades do Java para criar interfaces gráficas do usuário (*graphical user interfaces* — GUIs) e as imagens gráficas iniciais do livro antes das principais discussões mais profundas sobre esses tópicos mais adiante. Ele apresenta a tecnologia Swing madura do Java que, no momento em que esta obra era escrita, ainda era um pouco mais popular do que a nova tecnologia JavaFX mostrada nos capítulos posteriores.

O estudo de caso sobre GUI e imagens gráficas aparece em dez seções curtas (veja a Figura 3.11). Cada uma delas introduz vários novos conceitos e fornece exemplos de capturas de tela que mostram interações e resultados. Nas poucas primeiras seções, você criará seus primeiros aplicativos gráficos. Nas subsequentes, você utilizará conceitos da programação orientada a objetos para criar um aplicativo que desenha várias formas. Ao introduzirmos as GUIs formalmente no Capítulo 12, empregaremos o mouse para escolher exatamente que formas desenhar e onde as desenhar. No Capítulo 13, adicionaremos capacidades gráficas da API Java 2D para desenhar as formas com diferentes espessuras de linha e preenchimentos. Esperamos que este estudo de caso seja informativo e divertido para você.

Localização	Título — Exercício(s)
Seção 3.6	Utilizando caixas de diálogo — entrada e saída básicas com caixas de diálogo
Seção 4.15	Criando desenhos simples — exibindo e desenhando linhas na tela
Seção 5.11	Desenhando retângulos e ovais — utilizando formas para representar dados
Seção 6.13	Cores e formas preenchidas — desenhando um alvo e gráficos aleatórios
Seção 7.17	Desenhando arcos — desenhando espirais com arcos
Seção 8.16	Utilizando objetos com imagens gráficas — armazenando formas como objetos
Seção 9.7	Exibindo texto e imagens utilizando rótulos — fornecendo informações de status
Seção 10.11	Desenhando com polimorfismo — identificando as semelhanças entre as formas
Exercício 12.17	Expandindo a interface — utilizando componentes GUI e tratamento de evento
Exercício 13.31	Adicionando Java 2D — utilizando a API Java 2D para aprimorar desenhos

Figura 3.11 | Resumo da GUI e estudo de caso de imagens gráficas em cada capítulo.

Exibindo texto em uma caixa de diálogo

Os programas apresentados até agora exibem a saída na *janela de comando*. Muitos aplicativos utilizam janelas ou **caixas de diálogo** (também chamadas **diálogos**) para exibir a saída. Navegadores web como o Chrome, Firefox, Internet Explorer, Safari e Opera apresentam páginas da web em janelas próprias. Os programas de correio eletrônico permitem digitar e ler mensagens em uma janela. Tipicamente, as caixas de diálogo são janelas nas quais os programas mostram mensagens importantes aos usuários. A classe `JOptionPane` fornece caixas de diálogo pré-construídas que permitem aos programas exibir janelas que contêm mensagens — essas janelas são chamadas de **diálogos de mensagem**. A Figura 3.12 exibe a `String` `"Welcome to Java"` em um diálogo de mensagem.

```
1  // Figura 3.12: Dialog1.java
2  // Usando JOptionPane para exibir múltiplas linhas em uma caixa de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class Dialog1
6  {
7     public static void main(String[] args)
8     {
9        // exibe um diálogo com uma mensagem
10       JOptionPane.showMessageDialog(null, "Welcome to Java");
11    }
12 } // fim da classe Dialog1
```

Figura 3.12 | Utilizando `JOptionPane` para exibir múltiplas linhas em uma caixa de diálogo.

Classe JOptionPane, método static showMessageDialog

A linha 3 indica que o programa utiliza a classe `JOptionPane` do pacote **`javax.swing`**. Esse pacote contém muitas classes que ajudam a criar **GUIs** para aplicativos. **Componentes GUI** facilitam a entrada de dados executada pelo usuário de um programa e a apresentação das saídas a esse usuário. A linha 10 chama o método `JOptionPane` **`showMessageDialog`** para exibir uma caixa de diálogo que contém uma mensagem. O método requer dois argumentos. O primeiro ajuda o aplicativo Java a determinar onde posicionar a caixa de diálogo. Um diálogo é tipicamente exibido a partir de um aplicativo GUI em uma janela própria. O primeiro argumento refere-se a essa janela (conhecida como a *janela pai*) e faz o diálogo aparecer centralizado na janela do aplicativo. Se o primeiro argumento for `null`, a caixa de diálogo será exibida no centro da tela. O segundo argumento é a `String` a ser exibida na caixa de diálogo.

Introduzindo métodos static

O método `JOptionPane showMessageDialog` é o chamado **método `static`**. Tais métodos muitas vezes definem tarefas frequentemente adotadas. Por exemplo, muitos programas exibem caixas de diálogo, e o código para isso sempre é o mesmo. Em vez de exigir que você "reinvente a roda" e crie o código para exibir uma caixa de diálogo, os projetistas da classe `JOptionPane` declararam um método `static` que realiza essa tarefa para você. Um método `static` é chamado utilizando seu nome de classe seguido por um ponto (`.`) e o nome de método, como em

NomeDaClasse`.`*nomeDoMétodo*`(`*argumentos*`)`

Note que você *não* cria um objeto da classe `JOptionPane` para utilizar seu método `static showMessageDialog`. Discutiremos métodos `static` mais detalhadamente no Capítulo 6.

Inserindo texto em uma caixa de diálogo

A Figura 3.13 utiliza outra caixa de diálogo `JOptionPane` predefinida chamada **caixa de diálogo de entrada** que permite ao usuário *inserir dados* em um programa. O programa solicita o nome do usuário e responde com um diálogo de mensagem que contém uma saudação e o nome que o usuário inseriu.

```
1  // Figura 3.13: NameDialog.Java
2  // Obtendo a entrada de usuário a partir de um diálogo.
3  import javax.swing.JOptionPane;
4
5  public class NameDialog
6  {
7     public static void main(String[] args)
8     {
9        // pede para o usuário inserir seu nome
10       String name = JOptionPane.showInputDialog("What is your name?");
11
12       // cria a mensagem
13       String message =
14          String.format("Welcome, %s, to Java Programming!", name);
15
16       // exibe a mensagem para cumprimentar o usuário pelo nome
17       JOptionPane.showMessageDialog(null, message);
18    } // fim de main
19 } // termina NameDialog
```

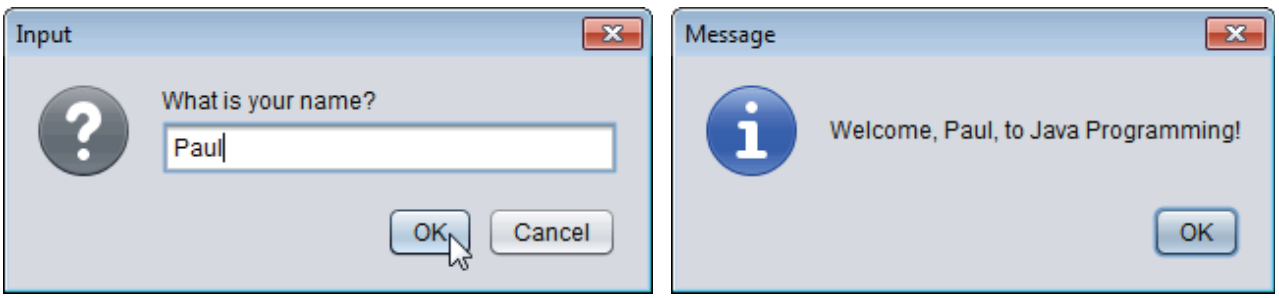


Figura 3.13 | Obtendo a entrada de usuário a partir de um diálogo.

Método `static` `showInputDialog` da classe `JOptionPane`

A linha 10 usa o método **`showInputDialog`** de `JOptionPane` para exibir uma caixa de diálogo de entrada que contém um prompt e um *campo* (conhecido como **campo de texto**) no qual o usuário pode inserir o texto. O argumento do método `showInputDialog` é o prompt que indica o nome que o usuário deve inserir. Ele digita caracteres no campo de texto, depois clica no botão OK ou pressiona a tecla *Enter* para retornar a `String` para o programa. O método `showInputDialog` retorna uma `String` contendo os caracteres digitados pelo usuário. Armazenamos a `String` na variável `name`. Se você pressionar o botão do diálogo Cancel ou a tecla *Esc*, o método retornará `null` e o programa exibirá a palavra "null" como nome.

Método `static` `format` da classe `String`

As linhas 13 e 14 utilizam o método `static String` **`format`** para retornar uma `String` que contém uma saudação com o nome do usuário. O método `format` funciona como `System.out.printf`, exceto que `format` retorna a `String` formatada em vez de exibi-la em uma janela de comando. A linha 17 mostra a saudação em uma caixa de diálogo de mensagem, assim como fizemos na Figura 3.12.

Exercício do estudo de caso GUI e imagens gráficas

3.1 Modifique o programa de adição na Figura 2.7 para utilizar entrada e saída baseadas em caixas de diálogo com os métodos da classe `JOptionPane`. Uma vez que o método `showInputDialog` retorna uma `String`, você deve converter a `String` que o usuário insere em um `int` para utilização em cálculos. O método `static parseInt` da classe `Integer` (pacote `java.lang`) recebe um argumento `String` que representa um inteiro e retorna o valor como um `int`. Se a `String` não contiver um inteiro válido, o programa terminará com um erro.

3.7 Conclusão

Neste capítulo, você aprendeu a criar suas próprias classes e métodos, criar objetos dessas classes e chamar métodos desses objetos para executar ações úteis. Você declarou variáveis de instância de uma classe a fim de manter os dados para cada objeto da classe e também seus próprios métodos para operar nesses dados. Aprendeu a chamar um método para instruí-lo a fazer sua tarefa e de que maneira passar informações para um método como argumentos cujos valores são atribuídos aos parâmetros dele. Descobriu a diferença entre variável local de um método e variável de instância de uma classe, e que apenas variáveis de instância são inicializadas automaticamente. Ainda, verificou como utilizar um construtor da classe para especificar os valores iniciais às variáveis de instância de um objeto. Você viu como criar diagramas de classe UML que modelam os métodos, atributos e construtores das classes. Por fim, você aprendeu a lidar com os números de ponto flutuante (números com separador decimal) — como armazená-los com variáveis de tipo primitivo `double`, como inseri-los com um objeto `Scanner` e como formatá-los com `printf` e o especificador `%f` para propósitos de exibição. [No Capítulo 8, começaremos representando valores monetários precisamente com a classe `BigDecimal`.] Também deve ter começado o estudo de caso opcional de GUI e imagens gráficas, aprendendo a escrever seus primeiros aplicativos GUI. No próximo capítulo iniciaremos nossa introdução às instruções de controle, que especificam a ordem em que as ações de um programa são realizadas. Você as utilizará em seus métodos para especificar como eles devem realizar suas tarefas.

Resumo

Seção 3.2 Variáveis de instância, métodos set *e métodos* get

- Cada classe que você cria torna-se um novo tipo que pode ser utilizado para declarar variáveis e elaborar objetos.
- Você pode declarar novos tipos de classe conforme necessário; essa é uma razão pela qual o Java é conhecido como uma linguagem extensível.

Seção 3.2.1 Classe `Account` com uma variável de instância, um método set *e um método* get

- Toda declaração de classe que inicia com o modificador de acesso `public` deve ser armazenada em um arquivo que tem o mesmo nome que a classe e termina com a extensão de arquivo `.java`.
- Cada declaração de classe contém a palavra-chave `class` seguida imediatamente do nome da classe.
- Os nomes de classe, método e variável são identificadores. Por convenção, todos usam nomes na notação camelo. Os nomes de classe começam com letra maiúscula, e os de método e variável, com uma letra minúscula.
- Um objeto tem atributos que são implementados como variáveis de instância que eles mantêm ao longo de sua vida.
- Existem variáveis de instância antes que os métodos sejam chamados em um objeto, enquanto os métodos são executados e depois que essa ação foi concluída.
- Uma classe normalmente contém um ou mais dos métodos que manipulam as variáveis de instância que pertencem a objetos específicos da classe.
- Variáveis de instância são declaradas dentro de uma declaração de classe, mas fora do corpo das instruções de método da classe.

- Cada objeto (instância) da classe tem sua própria cópia de cada uma das variáveis de instância da classe.
- A maioria das declarações de variável de instância é precedida pela palavra-chave `private`, que é um modificador de acesso. As variáveis ou métodos declarados com o modificador de acesso `private` só são acessíveis a métodos da classe em que são declarados.
- Os parâmetros são declarados em uma lista de itens separados por vírgula, que está localizada entre os parênteses que vêm depois do nome do método na declaração dele. Múltiplos parâmetros são separados por vírgulas. Cada parâmetro deve especificar um tipo seguido por um nome de variável.
- As variáveis declaradas no corpo de um método particular são conhecidas como locais e só podem ser utilizadas nesse método. Quando ele terminar, os valores de suas variáveis locais são perdidos. Os parâmetros de um método são variáveis locais dele.
- O corpo de todos os métodos é delimitado pelas chaves esquerda e direita (`{` e `}`).
- O corpo de cada método contém uma ou mais instruções que executam a(s) tarefa(s) desse método.
- O tipo de retorno do método especifica o tipo de dado retornado para o chamador de um método. A palavra-chave `void` indica que um método realizará uma tarefa, mas não retornará nenhuma informação.
- Os parênteses vazios que seguem um nome de método indicam que ele não requer nenhum parâmetro para realizar sua tarefa.
- Quando um método que especifica um tipo de retorno diferente de `void` for chamado e completar sua tarefa, ele retornará um resultado para seu método de chamada.
- A instrução `return` passa um valor a partir de um método chamado de volta para seu chamador.
- As classes costumam fornecer métodos `public` para permitir aos clientes da classe *set* (configurar) ou *get* (obter) variáveis de instância `private`. Os nomes desses métodos não precisam começar com *set* ou *get,* mas essa convenção de nomenclatura é recomendada.

Seção 3.2.2 Classe `AccountTest` que cria e usa um objeto da classe `Account`

- Uma classe que cria um objeto de outra classe, e então chama os métodos do objeto, é uma *driver*.
- O método `Scanner` `nextLine` lê os caracteres até um caractere de nova linha ser encontrado, depois retorna os caracteres como um método `String`.
- O método `Scanner` `next` lê os caracteres até qualquer um de espaço em branco ser encontrado, então retorna os caracteres como uma `String`.
- A expressão de criação de instância de classe começa com a palavra-chave `new` e estabelece um novo objeto.
- Um construtor é semelhante a um método, mas é chamado implicitamente pelo operador `new` para inicializar as variáveis de instância de um objeto no momento em que ele é criado.
- Para chamar um método de um objeto, o nome da variável deve ser seguido de um ponto separador, do nome de método e de um conjunto de parênteses que contém os argumentos do método.
- Variáveis locais não são inicializadas automaticamente. Cada variável de instância tem um valor inicial padrão — fornecido pelo Java quando você não especifica o valor inicial dela.
- O valor padrão para uma variável de instância do tipo `String` é `null`.
- Uma chamada de método fornece valores — conhecidos como argumentos — para cada um dos parâmetros dele. O valor de cada argumento é atribuído ao parâmetro correspondente no cabeçalho do método.
- O número de argumentos na chamada de método deve corresponder ao de itens na lista de parâmetros da declaração do método.
- Os tipos de argumento na chamada de método devem ser consistentes com os dos parâmetros correspondentes na declaração do método.

Seção 3.2.3 Compilação e execução de um aplicativo com múltiplas classes

- O comando `javac` pode compilar múltiplas classes ao mesmo tempo. Basta listar os nomes dos arquivos do código-fonte após o comando com cada um deles separado do próximo por um espaço. Se o diretório contendo o aplicativo incluir apenas os arquivos de um único aplicativo, você pode compilar todas as classes com o comando `javac *.java`. O asterisco (`*`) em `*.java` indica que todos os arquivos no diretório atual que têm a extensão de nome de arquivo "`.java`" devem ser compilados.

Seção 3.2.4 Diagrama de classe UML de `Account` com uma variável de instância e os métodos set *e* get

- Na UML, cada classe é modelada em um diagrama de classe como um retângulo com três compartimentos. Aquele na parte superior contém o nome da classe centralizado horizontalmente em negrito. O compartimento do meio exibe os atributos da classe, que correspondem às variáveis de instância em Java. O inferior inclui as operações da classe, que correspondem a métodos e construtores em Java.
- A UML representa variáveis de instância como um nome de atributo, seguido por dois-pontos e o tipo.
- Os atributos privados são precedidos por um sinal de subtração (–) na UML.
- A UML modela operações listando o nome delas seguido por um conjunto de parênteses. Um sinal de adição (+) na frente do nome da operação indica que é uma do tipo public na UML (isto é, um método `public` em Java).
- A UML modela um parâmetro de uma operação listando o nome dele, seguido por um caractere de dois-pontos e o tipo dele entre parênteses depois do nome de operação.
- A UML indica o tipo de retorno de uma operação colocando dois-pontos e ele depois dos parênteses que se seguem ao nome da operação.

- Os diagramas de classe UML não especificam tipos de retorno para operações que não retornam valores.
- Declarar variáveis de instância `private` é conhecido como ocultar dados ou informações.

Seção 3.2.5 Notas adicionais sobre a classe `AccountTest`

- Você deve chamar a maioria dos métodos, exceto `main`, explicitamente para instruí-los a fazer suas tarefas.
- Uma parte fundamental da ativação da JVM para localizar e chamar o método `main` a fim de começar a execução do aplicativo é a palavra-chave `static`, que indica que `main` é um método `static` que pode ser chamado sem antes criar um objeto da classe em que esse método é declarado.
- A maioria das outras classes que você utilizará nos programas Java precisa ser importada explicitamente. Há um relacionamento especial entre as classes que são compiladas no mesmo diretório. Por padrão, essas classes são consideradas como estando no mesmo pacote — conhecido como pacote padrão. As classes do mesmo pacote são importadas implicitamente para os arquivos de código-fonte de outras classes desse mesmo pacote. Uma declaração `import` não é necessária quando uma classe em um pacote utiliza outra no mesmo pacote.
- Uma declaração `import` não é necessária se você sempre se referir a uma classe com um nome totalmente qualificado, que inclui o nome do pacote e o nome da classe.

Seção 3.2.6 Engenharia de software com variáveis de instância `private` e métodos set *e* get `public`

- Declarar variáveis de instância `private` é conhecido como ocultar dados ou informações.

Seção 3.3 Tipos primitivos versus *tipos por referência*

- Tipos no Java são divididos em duas categorias — primitivos e por referência. Os tipos primitivos são `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` e `double`. Todos os outros são por referência; portanto, classes, que especificam os tipos de objeto, são tipos por referência.
- Uma variável de tipo primitivo pode armazenar exatamente um valor de seu tipo declarado por vez.
- As variáveis de instância de tipo primitivo são inicializadas por padrão. Variáveis dos tipos `byte`, `char`, `short`, `int`, `long`, `float` e `double` são inicializadas como `0`. As variáveis de tipo `boolean` são inicializadas como `false`.
- Variáveis de tipo por referência (chamadas referências) armazenam o local de um objeto na memória do computador. Essas variáveis referenciam objetos no programa. O objeto que é referenciado pode conter muitas variáveis de instância e métodos.
- As variáveis de instância de tipo por referência são inicializadas por padrão como valor `null`.
- Uma referência a um objeto é necessária para chamar os métodos de um objeto. Uma variável de tipo primitivo não referencia um objeto e, portanto, não pode ser utilizada para invocar um método.

Seção 3.4 Classe `Account`: inicialização de objetos com construtores

- Cada classe que você declara pode fornecer um construtor com parâmetros a ser utilizados para inicializar um objeto de uma classe quando ele for criado.
- O Java requer uma chamada de construtor para cada objeto que é criado.
- Construtores podem especificar parâmetros, mas não tipos de retorno.
- Se uma classe não definir construtores, o compilador fornecerá um construtor padrão sem parâmetros, e as variáveis de instância da classe serão inicializadas com seus valores padrão.
- Se você declarar um construtor para uma classe, o compilador *não* criará um *construtor padrão* para ela.
- A UML modela os construtores no terceiro compartimento de um diagrama de classe. Para distinguir entre um construtor e operações de uma classe, a UML coloca a palavra "constructor" entre aspas francesas (« e ») antes do nome do construtor.

Seção 3.5 A classe `Account` com um saldo; números de ponto flutuante

- Um número de ponto flutuante tem um ponto decimal. O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória — `float` e `double`.
- As variáveis de tipo `float` representam números de ponto flutuante de precisão simples e têm sete dígitos significativos. Já as variáveis de tipo `double` indicam números de ponto flutuante de dupla precisão. Elas requerem duas vezes a quantidade de memória das variáveis `float` e fornecem 15 dígitos significativos — aproximadamente o dobro da precisão de variáveis `float`.
- Literais de ponto flutuante são do tipo `double` por padrão.
- O método `nextDouble` de `Scanner` retorna um valor `double`.
- O especificador de formato `%f` é utilizado para gerar saída de valores de tipo `float` ou `double`. Já o especificador de formato `%.2f` especifica que dois dígitos da precisão devem ser gerados à direita do ponto decimal no número de ponto flutuante.
- O valor padrão para uma variável de instância do tipo `double` é `0.0`, e o valor padrão para uma variável de instância do tipo `int` é `0`.

Exercícios de revisão

3.1 Preencha as lacunas em cada uma das seguintes sentenças:

a) Toda declaração de classe que inicia com a palavra-chave _____ deve ser armazenada em um arquivo que tem exatamente o mesmo nome que a classe e terminar com a extensão de nome do arquivo `.java`.

b) A palavra-chave _____ em uma declaração de classe é imediatamente seguida pelo nome da classe.

c) A palavra-chave _____ solicita memória do sistema para armazenar um objeto, e então chama o construtor da classe correspondente para inicializar esse objeto.

d) Todo parâmetro deve especificar um(a) _____ e um(a) _____.

e) Por padrão, as classes que são compiladas no mesmo diretório são consideradas como estando no mesmo pacote, conhecido como _____.

f) O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória: _____ e _____.

g) As variáveis de tipo `double` representam números de ponto flutuante de _____.

h) O método `scanner` _____ retorna um valor `double`.

i) A palavra-chave `public` é um _____ de acesso.

j) O tipo de retorno _____ indica que um método não retornará um valor.

k) O método `Scanner` _____ lê os caracteres até encontrar um caractere de nova linha, então retorna esses caracteres como uma `String`.

l) A classe `String` está no pacote _____.

m) Um(a) _____ não é requerido(a) se você sempre referenciar uma classe por meio do seu nome completamente qualificado.

n) Um(a) _____ é um número com um ponto de fração decimal, como 7,33, 0,0975 ou 1000,12345.

o) As variáveis de tipo `float` representam _____ números de ponto flutuante de dupla precisão.

p) O especificador de formato _____ é utilizado para gerar saída de valores de tipo `float` ou `double`.

q) Os tipos no Java são divididos em duas categorias — tipo _____ e tipo _____.

3.2 Determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Por convenção, os nomes de método são iniciados com letra maiúscula, e todas as palavras subsequentes a ele também começam com letra maiúscula.

b) Uma declaração `import` não é necessária quando uma classe em um pacote utiliza outra no mesmo pacote.

c) Parênteses vazios que se seguem a um nome de método em uma declaração indicam que ele não requer nenhum parâmetro para realizar sua tarefa.

d) Uma variável de tipo primitivo pode ser utilizada para invocar um método.

e) As variáveis declaradas no corpo de um método particular são conhecidas como variáveis de instância e podem ser utilizadas em todos os métodos da classe.

f) O corpo de todos os métodos é delimitado pelas chaves esquerda e direita (`{` e `}`).

g) As variáveis locais de tipo primitivo são inicializadas por padrão.

h) As variáveis de instância de tipo por referência são inicializadas por padrão com o valor `null`.

i) Qualquer classe que contém `public static void main(String[] args)` pode ser usada para executar um aplicativo.

j) O número de argumentos na chamada de método deve corresponder ao de itens na lista de parâmetros da declaração desse método.

k) Os valores de ponto flutuante que aparecem no código-fonte são conhecidos como literais de ponto flutuante e são tipos `float` por padrão.

3.3 Qual é a diferença entre uma variável local e uma variável de instância?

3.4 Explique o propósito de um parâmetro de método. Qual a diferença entre um parâmetro e um argumento?

Respostas dos exercícios de revisão

3.1 a) `public`. b) `class`. c) `new`. d) tipo, nome. e) pacote padrão. f) `float`, `double`. g) precisão dupla. h) `nextDouble`. i) modificador. j) `void`. k) `nextLine`. l) `java.lang`. m) declaração `import`. n) número de ponto flutuante. o) simples. p) `%f`. q) primitivo, referência.

3.2 a) Falsa. Por convenção, os nomes de método são iniciados com letra minúscula e todas as palavras subsequentes começam com letra maiúscula. b) Verdadeira. c) Verdadeira. d) Falsa. Uma variável de tipo primitivo não pode ser utilizada para invocar um método — uma referência a um objeto é necessária para que os métodos do objeto possam ser invocados. e) Falsa. Essas variáveis são chamadas variáveis locais e só podem ser utilizadas no método em que são declaradas. f) Verdadeira. g) Falsa. As variáveis de instância de tipo primitivo são inicializadas por padrão. Deve-se atribuir um valor explicitamente a cada variável local. h) Verdadeira. i) Verdadeira. j) Verdadeira. k) Falsa. Esses literais são de tipo `double` por padrão.

3.3 Uma variável local é declarada no corpo de um método e só pode ser utilizada do ponto em que isso acontece até o fim da declaração do método. Uma variável de instância é declarada em uma classe, mas não no corpo de qualquer um dos métodos dessa classe. Além disso, as variáveis de instância são acessíveis a todos os métodos da classe. (Veremos uma exceção disso no Capítulo 8.)

3.4 Um parâmetro representa informações adicionais que um método requer para realizar sua tarefa. Cada parâmetro requerido por um método é especificado na declaração do método. Um argumento é o valor real de um parâmetro de método. Quando um método é chamado, os valores de argumento são passados para os parâmetros correspondentes desse método para que ele possa realizar sua tarefa.

Questões

3.5 (***Palavra-chave new***) Qual é o objetivo da palavra-chave `new`? Explique o que acontece quando você a utiliza.

3.6 (***Construtores padrão***) O que é um construtor padrão? Como as variáveis de instância de um objeto são inicializadas se uma classe tiver somente um construtor padrão?

3.7 (***Variáveis de instância***) Explique o propósito de uma variável de instância.

3.8 (***Usando classes sem importá-las***) A maioria das classes precisa ser importada antes de ser usada em um aplicativo. Por que cada aplicativo pode utilizar as classes `System` e `String` sem importá-las antes?

3.9 (***Usando uma classe sem importá-la***) Explique como um programa pode usar a classe `Scanner` sem importá-la.

3.10 (***Métodos* set *e* get**) Explique por que uma classe pode fornecer um método *set* e um método *get* para uma variável de instância.

3.11 (***Classe `Account` modificada***) Modifique a classe `Account` (Figura 3.8) para fornecer um método chamado `withdraw` que retira dinheiro de uma `Account`. Assegure que o valor de débito não exceda o saldo de `Account`. Se exceder, o saldo deve ser deixado inalterado e o método deve imprimir uma mensagem que indica `"Withdrawal amount exceeded account balance"` [`Valor de débito excedeu o saldo da conta`]. Modifique a classe `AccountTest` (Figura 3.9) para testar o método `withdraw`.

3.12 (***Classe `Invoice`***) Crie uma classe chamada `Invoice` para que uma loja de suprimentos de informática a utilize para representar uma fatura de um item vendido nela. Uma `Invoice` (fatura) deve incluir quatro partes das informações como variáveis de instância — o número (tipo `String`), a descrição (tipo `String`), a quantidade comprada de um item (tipo `int`) e o preço por item (`double`). Sua classe deve ter um construtor que inicializa as quatro variáveis de instância. Forneça um método *set* e um *get* para cada variável de instância. Além disso, forneça um método chamado `getInvoiceAmount` que calcula o valor de fatura (isto é, multiplica a quantidade pelo preço por item) e depois retorna esse valor como `double`. Se a quantidade não for positiva, ela deve ser configurada como `0`. Se o preço por item não for positivo, ele deve ser configurado como `0.0`. Escreva um aplicativo de teste chamado `InvoiceTest` que demonstra as capacidades da classe `Invoice`.

3.13 (***Classe `Employee`***) Crie uma classe chamada `Employee` que inclua três variáveis de instância — um primeiro nome (tipo `String`), um sobrenome (tipo `String`) e um salário mensal (`double`). Forneça um construtor que inicializa as três variáveis de instância. Forneça um método *set* e um *get* para cada variável de instância. Se o salário mensal não for positivo, não configure seu valor. Escreva um aplicativo de teste chamado `EmployeeTest` que demonstre as capacidades da classe `Employee`. Crie dois objetos `Employee` e exiba o salário *anual* de cada objeto. Então dê a cada `Employee` um aumento de 10% e exiba novamente o salário anual de cada `Employee`.

3.14 (***Classe `Date`***) Crie uma classe chamada `Date` que inclua três variáveis de instância — mês (tipo `int`), dia (tipo `int`) e ano (tipo `int`). Forneça um construtor que inicializa as três variáveis de instância supondo que os valores fornecidos estejam corretos. Ofereça um método *set* e um *get* para cada variável de instância. Apresente um método `displayDate` que exiba mês, dia e ano separados por barras normais (`/`). Escreva um aplicativo de teste chamado `DateTest` que demonstre as capacidades da classe `Date`.

3.15 (***Removendo código duplicado no método* `main`**) Na classe `AccountTest` da Figura 3.9, o método `main` contém seis instruções (linhas 13 e 14, 15 e 16, 28 e 29, 30 e 31, 40 e 41 e 42 e 43) e cada uma exibe `name` e `balance` do objeto `Account`. Estude essas instruções e você perceberá que elas só diferem no objeto `Account` sendo manipulado — `account1` ou `account2`. Neste exercício, você definirá um novo método `displayAccount` que contém *uma* cópia dessa instrução de saída. O parâmetro do método será um objeto `Account` e o método irá gerar `name` e `balance` dele. Então você substituirá as seis instruções duplicadas em `main` por chamadas para `displayAccount` passando como argumento o objeto específico `Account` para saída.

Modifique a classe `AccountTest` da Figura 3.9 para declarar o seguinte método `displayAccount` *após* a chave direita de fechamento de `main` e *antes* da chave direita de fechamento da classe `AccountTest`:

```
public static void displayAccount(Account accountToDisplay)
{
   // coloque aqui a instrução que exibe
   // o name e o balance de accountToDisplay
}
```

Substitua o comentário no corpo do método por uma instrução que exiba `name` e `balance` de `accountToDisplay`.

Lembre-se de que `main` é um método `static`, assim pode ser chamado sem antes criar um objeto da classe em que é declarado. Também declaramos o método `displayAccount` como um método `static`. Quando `main` tem de chamar outro método na mesma classe sem antes criar um objeto dela, o outro método *também* deve ser declarado `static`.

Depois de concluir a declaração de `displayAccount`, modifique `main` para substituir as instruções que exibem `name` e `balance` de cada `Account` pelas chamadas para `displayAccount` — cada uma recebendo como seu argumento o objeto `account1` ou `account2`, como apropriado. Então, teste a classe `AccountTest` atualizada para garantir que ela produz a mesma saída como mostrado na Figura 3.9.

Fazendo a diferença

3.16 **(*Calculadora de frequência cardíaca alvo*)** Ao fazer exercícios físicos, você pode utilizar um monitor de frequência cardíaca para ver se sua frequência permanece dentro de um intervalo seguro sugerido pelos seus treinadores e médicos. Segundo a American Heart Association (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), a fórmula para calcular a *frequência cardíaca máxima* por minuto é 220 menos a idade em anos. Sua *frequência cardíaca alvo* é um intervalo entre 50-85% da sua frequência cardíaca máxima. [***Observação:*** essas fórmulas são estimativas fornecidas pela AHA. As frequências cardíacas máximas e alvo podem variar com base na saúde, capacidade física e sexo da pessoa. **Sempre consulte um médico ou profissional de saúde qualificado antes de começar ou modificar um programa de exercícios físicos.**] Crie uma classe chamada `HeartRates`. Os atributos da classe devem incluir o nome, sobrenome e data de nascimento da pessoa (consistindo em atributos separados para mês, dia e ano de nascimento). Sua classe deve ter um construtor que receba esses dados como parâmetros. Para cada atributo forneça métodos *set* e *get*. A classe também deve incluir um método que calcule e retorne a idade (em anos), um que calcule e retorne a frequência cardíaca máxima e um que calcule e retorne a frequência cardíaca alvo da pessoa. Escreva um aplicativo Java que solicite as informações da pessoa, instancie um objeto da classe `HeartRates` e imprima as informações a partir desse objeto — incluindo nome, sobrenome e data de nascimento da pessoa — calcule e imprima a idade da pessoa (em anos), seu intervalo de frequência cardíaca máxima e sua frequência cardíaca alvo.

3.17 **(*Computadorização dos registros de saúde*)** Uma questão relacionada à assistência médica discutida ultimamente nos veículos de comunicação é a computadorização dos registros de saúde. Essa possibilidade está sendo abordada de maneira cautelosa por causa de preocupações quanto à privacidade e à segurança de dados sigilosos, entre outros motivos. [Iremos discutir essas preocupações em exercícios posteriores.] A computadorização dos registros de saúde pode facilitar que pacientes compartilhem seus perfis e históricos de saúde entre vários profissionais de saúde. Isso talvez aprimore a qualidade da assistência médica, ajude a evitar conflitos e prescrições erradas de medicamentos, reduza custos em ambulatórios e salve vidas. Neste exercício, você projetará uma classe `HealthProfile` "inicial" para uma pessoa. Os atributos da classe devem incluir nome, sobrenome, sexo, data de nascimento (consistindo em atributos separados para mês, dia e ano de nascimento), altura (em metros) e peso (em quilogramas) da pessoa. Sua classe deve ter um construtor que receba esses dados. Para cada atributo, forneça métodos *set* e *get*. A classe também deve incluir métodos que calculem e retornem a idade do usuário em anos, intervalo de frequência cardíaca máxima e frequência cardíaca alvo (veja o Exercício 3.16), além de índice de massa corporal (IMC; veja o Exercício 2.33). Escreva um aplicativo Java que solicite as informações da pessoa, instancie um objeto da classe `HealthProfile` para ela e imprima as informações a partir desse objeto — incluindo nome, sobrenome, sexo, data de nascimento, altura e peso da pessoa —, e então calcule e imprima a idade em anos, IMC, intervalo de frequência cardíaca máxima e frequência cardíaca alvo. Ele também deve exibir o gráfico de valores IMC do Exercício 2.33.