Capítulo 10
Programação orientada a objetos: polimorfismo

Java™ Como Programar, 8/E

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

OBJETIVOS

Neste capítulo, você aprenderá:

- O conceito de polimorfismo.
- A utilizar métodos sobrescritos para executar o polimorfismo.
- A distinguir entre classes concretas e abstratas.
- A declarar métodos abstratos para criar classes abstratas.
- Como o polimorfismo torna sistemas extensíveis e sustentáveis.
- A determinar um tipo de objeto em tempo de execução.
- A declarar e implementar interfaces.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.1 Introdução

- **Polimorfismo**

  Permite "programar no geral" em vez de "programar no específico".

  O polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse como se todas fossem objetos da superclasse; isso pode simplificar a programação.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Exemplo: Suponha que criamos um programa que simula o movimento de vários tipos de animais para um estudo biológico. As classes `Peixe`, `Anfíbio` e `Pássaro` representam os três tipos de animais sob investigação.

  Cada classe estende a superclasse `Animal`, que contém um método `mover` e mantém a localização atual de um animal como coordenadas *x*-*y*. Toda subclasse implementa o método `mover`.

  Um programa mantém um array `Animal` que contém referências a objetos das várias subclasses `Animal`. Para simular os movimentos dos animais, o programa envia a mesma mensagem a cada objeto uma vez por segundo — a saber, `move`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Cada tipo específico de `Animal` responde a uma mensagem `mover` de uma maneira única:
  um `Peixe` poderia nadar um metro
  um `Sapo` poderia pular um metro e meio
  um `Pássaro` poderia voar três metros.
- O programa emite a mesma mensagem (isto é, `mover`) para cada objeto animal, mas cada objeto sabe como modificar suas coordenadas x-y apropriadamente de acordo com seu tipo específico de movimento.
- Contar com o fato de que cada objeto sabe como "agir corretamente" em resposta à mesma chamada de método é o conceito-chave do polimorfismo.
- A mesma mensagem enviada a uma variedade de objetos tem "muitas formas" de resultados — daí o termo polimorfismo.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Com o polimorfismo, podemos projetar e implementar sistemas que são facilmente *extensíveis.*

  Novas classes podem ser adicionadas com pouca ou nenhuma modificação a partes gerais do programa, contanto que as novas classes sejam parte da hierarquia de herança que o programa processa genericamente.

  As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que adicionamos à hierarquia.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Depois que uma classe implementa uma interface, todos os objetos dessa classe têm um relacionamento *é um* com o tipo de interface e temos a garantia de que todos os objetos da classe fornecem a funcionalidade descrita pela interface.
- Isso também é verdade para todas as subclasses dessa classe.
- Interfaces são particularmente úteis para atribuir funcionalidades comuns a classes possivelmente não-relacionadas.

  Permite que objetos de classes não relacionadas sejam processados polimorficamente — objetos de classes que implementam a mesma interface podem responder à todas as chamadas de método da interface.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Uma interface descreve um conjunto de métodos que pode ser chamado em um objeto, mas não fornece implementações concretas para todos os métodos.
- Você pode declarar classes que **implementam** (isto é, fornecem implementações concretas para os métodos de) uma ou mais interfaces.
- Cada método de interface deve ser declarado em todas as classes que implementam explicitamente a interface.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.2 Exemplos de polimorfismo

- Exemplo: Quadriláteros

  Se a classe `Retângulo` for derivada da classe `Quadrilátero`, então um objeto `Retângulo` é uma versão mais específica de um `Quadrilátero`.

  Qualquer operação realizada em um `Quadrilátero` também pode ser executada em um `Retângulo`.

  Essas operações também podem ser realizadas em outros `Quadriláteros`, como `Quadrados`, `Paralelogramas` e `Trapezoides`.

  O polimorfismo ocorre quando um programa invoca um método por meio de uma variável da superclasse `Quadrilátero`, a versão correta de subclasse do método é chamada com base no tipo da referência armazenada na variável da superclasse.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Exemplo: Objetos espaciais em um videogame

  Um videogame que manipula objetos das classes `Marciano`, `Venusiano`, `Plutoniano`, `NaveEspacial` e `CanhaoDeLaser`. Cada uma estende `ObjetoEspacial` e redefine seu método `draw`.

  Um programa de gerenciamento de tela mantém uma coleção de referências a objetos das várias classes e periodicamente envia a cada objeto a mesma mensagem, isto é, `desenhar`.

  Cada objeto, porém, responde de uma maneira única.

  Um objeto `Marciano` desenharia-se em vermelho com olhos verdes e o número apropriado de antenas.

  Um objeto `NaveEspacial` desenharia-se como disco voador prateado e brilhante.

  Um objeto `CanhãoDeLaser` poderia se desenhar como um feixe vermelho brilhante atravessando a tela.

  Mais uma vez, a mesma mensagem (nesse caso, `desenhar`) enviada a uma variedade de objetos tem "muitas formas" de resultados.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Um gerenciador de tela poderia utilizar o polimorfismo para facilitar a adição de novas classes a um sistema com modificações mínimas no código do sistema.
- Para adicionar novos objetos ao nosso videogame:

  Crie uma classe que estende `SpaceObject` e fornece sua própria implementação do método `draw`.

  Quando objetos dessa classe aparecem na coleção `ObjetoEspacial`, o código do gerenciador de tela invoca o método `desenhar`, exatamente como faz todos os outros objetos na coleção, independentemente do seu tipo.

  Portanto, os novos objetos são simplesmente "conectados" sem nenhuma modificação no código do gerenciador de tela pelo programador.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 10.1

O polimorfismo permite-lhe tratar as generalidades e deixar que o ambiente de tempo de execução trate as especificidades. Você pode instruir objetos a se comportarem de maneiras apropriadas para esses objetos, sem nem mesmo conhecer seus tipos (contanto que os objetos pertençam à mesma hierarquia de herança).

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Observação de engenharia de software 10.2

O polimorfismo promove extensibilidade: o software que invoca o comportamento polimórfico é independente dos tipos de objeto para os quais as mensagens são enviadas. Novos tipos de objetos que podem responder a chamadas de método existentes podem ser incorporados a um sistema sem modificar o sistema básico. Somente o código de cliente que instancia os novos objetos deve ser modificado para acomodar os novos tipos.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.3 Demonstrando um comportamento polimórfico

- No próximo exemplo, temos por alvo uma referência de superclasse em um objeto de subclasse.

  Invocar um método em um objeto de subclasse via uma referência de superclasse invoca as funcionalidades da subclasse

  O tipo do objeto referenciado, não o tipo de variável, é que determina qual método é chamado.
- Esse exemplo demonstra que um objeto de uma subclasse pode ser tratado como um objeto de sua superclasse, permitindo várias manipulações interessantes.
- Um programa pode criar um array de variáveis de superclasse que referencia objetos de muitos tipos de subclasse.

  Isso é permitido porque cada objeto de subclasse *é um* objeto da sua superclasse.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Um objeto de superclasse não pode ser tratado como um objeto de subclasse, porque um objeto de superclasse *não* é um objeto de nenhuma das suas subclasses.
- O relacionamento *é um* se aplica somente para cima na hierarquia, de uma subclasse para suas superclasses diretas (e indiretas), e não para baixo.
- O compilador Java *permite* a atribuição de uma referência de superclasse a uma variável de subclasse se você explicitamente fizer a coerção (casting) da referência de superclasse para o tipo de subclasse

  Uma técnica conhecida como **downcasting** permite ao programa invocar métodos de subclasse que não estão na superclasse.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
1  // Figura 10.1: PolymorphismTest.java
2  // Atribuindo referências de superclasse e subclasse a
3  // variáveis de superclasse e de subclasse.
4
5  public class PolymorphismTest
6  {
7     public static void main( String[] args )
8     {
9        // atribui uma referência de superclasse a variável de superclasse
10       CommissionEmployee commissionEmployee = new CommissionEmployee(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // atribui uma referência de subclasse a variável de subclasse
14       BasePlusCommissionEmployee basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoca toString no objeto de superclasse utilizando a variável de superclasse
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Call CommissionEmployee's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
```

A variável referencia um objeto `ComissionEmployee`, portanto o método `toString` dessa classe é chamado

Figura 10.1 | Atribuindo referências de superclasse e subclasse a variáveis de superclasse e subclasse. (Parte 1 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
23          // invoca toString no objeto de subclasse utilizando a variável de subclasse
24          System.out.printf( "%s %s:\n\n%s\n\n",
25             "Call BasePlusCommissionEmployee's toString with subclass",
26             "reference to subclass object",
27             basePlusCommissionEmployee.toString() );
28
29          // invoca toString no objeto de subclasse utilizando a variável de superclasse
30          CommissionEmployee commissionEmployee2 =
31             basePlusCommissionEmployee;
32          System.out.printf( "%s %s:\n\n%s\n",
33             "Call BasePlusCommissionEmployee's toString with superclass",
34       "reference to subclass object", commissionEmployee2.toString() );
35       } // fim de main
36    } // fim da classe PolymorphismTest
```

A variável referencia um objeto BasePlusComissionEmployee, portanto o método toString dessa classe é chamado

A variável referencia um objeto BasePlusComissionEmployee, portanto o método toString dessa classe é chamado

```
Call CommissionEmployee's toString with superclass reference to superclass object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

Figura 10.1 | Atribuindo referências de superclasse e subclasse a variáveis de superclasse e subclasse. (Parte 2 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
Call CommissionEmployee's toString with superclass reference to superclass object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Figura 10.1 | Atribuindo referências de superclasse e subclasse a variáveis de superclasse e subclasse. (Parte 3 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Quando uma variável de superclasse contém uma referência a um objeto de subclasse, e essa referência é utilizada para chamar um método, a versão de subclasse do método é chamada.

  O compilador Java permite esse "cruzamento" porque um objeto de uma subclasse *é um objeto da sua superclasse (mas não vice-versa).*
- Quando o compilador encontra uma chamada de método feita por meio de uma variável, ele determina se o método pode ser chamado verificando o tipo de classe da variável.

  Se essa classe contiver a declaração de método apropriada (ou estender uma), a chamada é compilada.
- Em tempo de execução, o tipo do objeto que a variável referencia determina o método real a utilizar.

  Esse processo é chamado de vinculação dinâmica.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.4 Classes e métodos abstratos

- **Classes abstratas**

  Às vezes é útil declarar classes para as quais você nunca pretende criar objetos.

  Utilizadas somente como superclasses em hierarquias de herança e, por isso, às vezes são chamadas de **superclasses abstratas.**

  As classes abstratas não podem ser utilizadas para instanciar objetos, porque são incompletas.

  As subclasses devem declarar as "partes ausentes" para tornarem-se classes "concretas", a partir das quais você pode instanciar objetos; caso contrário, essas subclasses, também, serão abstratas.
- Uma classe abstrata fornece uma superclasse que outras classes podem estender e, portanto, podem compartilhar um design comum.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- As classes que podem ser utilizadas para instanciar objetos são chamadas **classes concretas.**
- Essas classes fornecem implementações de cada método que elas declaram (algumas implementações podem ser herdadas).
- Superclasses abstratas são excessivamente gerais para criar objetos reais — elas só especificam o que é comum entre subclasses.
- As classes concretas fornecem os aspectos específicos que tornam razoável instanciar objetos.
- Nem todas as hierarquias contêm classes abstratas.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Programadores costumam escrevem código de cliente que utiliza apenas tipos abstratos de superclasse para reduzir dependências do código de cliente em um intervalo de tipos de subclasse.

  Você pode escrever um método com o parâmetro de um tipo de superclasse abstrata.

  Quando chamado, esse método pode receber um objeto de qualquer classe concreta que direta ou indiretamente estende a superclasse especificada como o tipo do parâmetro.
- As classes abstratas às vezes constituem vários níveis da hierarquia.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Você cria uma classe abstrata declarando-a com a palavra-chave **abstract.**
- Uma classe abstrata normalmente contém um ou mais **métodos abstratos.**
  Um método abstrato é aquele com a palavra-chave `abstract` na sua declaração, como em

```
public abstract void draw(); // método abstrato
```

- Métodos abstratos não fornecem implementações.
- Uma classe que contém métodos abstratos deve ser declarada mesmo que essa classe contenha alguns métodos concretos (não-abstratos).
- Cada subclasse concreta de uma superclasse abstrata também deve fornecer implementações concretas de cada um dos métodos abstratos da superclasse.
- Os construtores e métodos `static` não podem ser declarados `abstract`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Observação de engenharia de software 10.3

Uma classe abstrata declara atributos e comportamentos comuns (ambos abstratos e concretos) das várias classes em uma hierarquia de classes. Em geral, uma classe abstrata contém um ou mais métodos abstratos que as subclasses devem sobrescrever se elas precisarem ser concretas. Variáveis de instância e métodos concretos de uma classe abstrata estão sujeitos às regras normais da herança.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 10.1

Tentar instanciar um objeto de uma classe abstrata é um erro de compilação.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Erro comum de programação 10.2

A falha em implementar os métodos abstratos de uma superclasse em uma subclasse é um erro de compilação a menos que a subclasse também seja declarada `abstract`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Não é possível instanciar objetos de superclasses abstratas, mas você pode utilizar superclasses abstratas para declarar variáveis

  Estas podem armazenar referências a objetos de qualquer classe concreta derivada desses superclasses abstratas.

  Os programas em geral utilizam essas variáveis para manipular objetos de subclasse polimorficamente.
- É possível utilizar nomes abstratos de superclasse para invocar métodos `static` declarados nessas superclasses abstratas.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- O polimorfismo é particularmente eficaz para implementar os chamados *sistemas de software em camadas.*
- Exemplo: Sistemas operacionais e drivers de dispositivo.

  Comandos para ler (read) ou gravar (write) os dados de e a partir de dispositivos poderiam ter certa uniformidade.

  Drivers de dispositivo controlam toda comunicação entre o sistema operacional e os dispositivos.

  A mensagem de gravação enviada a um driver de dispositivo precisa ser interpretada especificamente no contexto desse driver e como ela manipula dispositivos de um tipo específico.

  A própria chamada de gravação não é realmente diferente de gravar em qualquer outro dispositivo no sistema — simplesmente transferir um número de bytes da memória para esse dispositivo.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Um sistema operacional orientado a objetos talvez utilize uma superclasse abstrata para fornecer uma "interface" apropriada para todos os drivers de dispositivo.
  Subclasses são formadas de modo a comportar-se de maneira semelhante.
  Os métodos do driver de dispositivo são declarados como métodos abstratos na superclasse abstrata.
  As implementações desses métodos abstratos são fornecidas nas subclasses que correspondem com os tipos de drivers de dispositivo específicos.
- Novos dispositivos são continuamente desenvolvidos.
  Ao comprar um novo dispositivo, ele vem com um driver de dispositivo fornecido pelo fornecedor do dispositivo e o dispositivo torna-se imediatamente operacional depois que você o conecta e instala o driver no seu computador.
- Esse é outro exemplo elegante de como o polimorfismo torna sistemas extensíveis.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5 Estudo de caso: sistema de folha de pagamento utilizando polimorfismo

- Utilize um método abstrato e o polimorfismo para realizar cálculos da folha de pagamento com base no tipo de hierarquia de herança de um funcionário.
- Requisitos da hierarquia de herança de empregados aprimorada:
  Uma empresa paga seus funcionários semanalmente. Os funcionários são de quatro tipos: Funcionários assalariados recebem salários fixos semanais independentemente do número de horas trabalhadas, funcionários que trabalham por hora são pagos da mesma forma e recebem horas extras (isto é, 1,5 vezes sua taxa de salário por hora) por todas as horas trabalhadas além das 40 horas normais, funcionários comissionados recebem uma porcentagem sobre suas vendas e funcionários assalariados/comissionados recebem um salário-base mais uma porcentagem sobre suas vendas. Para o período salarial atual, a empresa decidiu recompensar os funcionários assalariados/comissionados adicionando 10% aos seus salários-base. A empresa quer escrever um aplicativo Java que realiza os cálculos da folha de pagamento polimorficamente.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Utilizamos a classe `abstract Employee` para representar o conceito geral de um funcionário.
- Subclasses: `SalariedEmployee`, `CommissionEmployee`, `HourlyEmployee` e `BasePlusCommissionEmployee` (uma subclasse indireta)
- A Figura 10.2 mostra a hierarquia de herança do nosso aplicativo polimórfico de folha de pagamento de funcionários.
- Os nomes de classe abstrata são grafados em itálico na UML.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

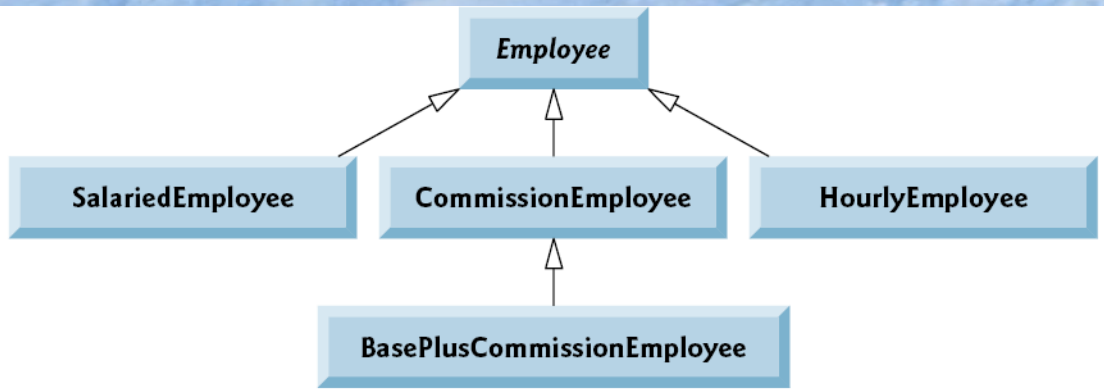

Figura 10.2 | Diagrama de classe da UML da hierarquia `Employee`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- A superclasse abstrata `Employee` declara a "interface" para a hierarquia — isto é, o conjunto de métodos que um programa pode invocar em todos os objetos `Employee`.

  Aqui, utilizamos o termo "interface" em um sentido geral para nos referirmos às várias maneiras como os programas se comunicam com objetos de qualquer subclasse `Employee`.
- Cada funcionário tem um nome, um sobrenome e um número de seguro social, definidos na superclasse `Employee`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5.1 Superclasse abstrata Employee

- A classe `Employee` (Figura 10.4) fornece métodos `earnings` e `toString`, além dos métodos *get* e *set* que manipulam as variáveis de instância de `Employee`.
- Um método `earnings` se aplica genericamente a todos os empregados, mas todos os cálculos de rendimentos dependem da classe do empregado.

  Um método `abstract` — não há informações suficientes para determinar que quantia `earnings` deve retornar.

  Cada subclasse sobrescreve `earnings` com uma implementação apropriada.
- Itera pelos elementos de `Employee`s e chama o método `earnings` para cada objeto de subclasse `Employee`.

  Chamadas de método processadas polimorficamente.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- O diagrama na Fig. 10.3 mostra cada uma das cinco classes na hierarquia no canto inferior esquerdo e os métodos `earnings` e `toString` na parte superior.
- Para cada classe, o diagrama mostra os resultados desejados de cada método.
- Declarar o método `earnings` como `abstract` indica que cada subclasse concreta deve fornecer uma implementação de `earnings` apropriada e que um programa será capaz de utilizar as variáveis da superclasse `Employee` para invocar o método `earnings` polimorficamente para o tipo `Employee`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Figura 10.3 | Interface polimórfica para as classes na hierarquia `Employee`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
1  // Figura 10.4: Employee.java
2  // Superclasse abstrata Employee.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // construtor com três argumentos
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // fim do construtor Employee com três argumentos
17
18    // configura o nome
19    public void setFirstName( String first )
20    {
21       firstName = first; // deve validar
22    } // fim do método setFirstName
23
```

Figura 10.4 | Superclasse `Employee` abstrata. (Parte 1 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
24       // retorna o nome
25       public String getFirstName()
26       {
27          return firstName;
28       } // fim do método getFirstName
29
30       // configura o sobrenome
31       public void setLastName( String last )
32       {
33          lastName = last; // deve validar
34       } // fim do método setLastName
35
36       // retorna o sobrenome
37       public String getLastName()
38       {
39          return lastName;
40       } // fim do método getLastName
41
42       // configura o CIC
43       public void setSocialSecurityNumber( String ssn )
44       {
45          socialSecurityNumber = ssn; // deve validar
46       } // fim do método setSocialSecurityNumber
47
```

Figura 10.4 | Superclasse `Employee` abstrata. (Parte 2 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
48       // retorna o CIC
49       public String getSocialSecurityNumber()
50       {
51          return socialSecurityNumber;
52       } // fim do método getSocialSecurityNumber
53
54       // retorna a representação de String do objeto Employee
55       @Override
56       public String toString()
57       {
58          return String.format( "%s %s\nsocial security number: %s",
59             getFirstName(), getLastName(), getSocialSecurityNumber() );
60       } // fim do método toString
61
62       // método abstract sobrescrito por subclasses concretas
63       public abstract double earnings(); // nenhuma implementação aqui
64    } // fim da classe abstrata Employee
```

Esse método deve ser sobrescrito nas subclasses para torná-las concretas

Figura 10.4 | Superclasse `Employee` abstrata. (Parte 3 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5.2 Subclasse concreta SalariedEmployee

```
1  // Figura 10.5: SalariedEmployee.java
2  // A classe concreta SalariedEmployee estende a classe Employee abstrata.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // construtor com quatro argumentos
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // passa para o construtor Employee
13       setWeeklySalary( salary ); // valida e armazena o salário
14    } // fim do construtor SalariedEmployee com quatro argumentos
15
16    // configura o salário
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // fim do método setWeeklySalary
21
```

Figura 10.5 | A classe concreta `SalariedEmployee` estende a classe `abstract` `Employee`. (Parte 1 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
22       // retorna o salário
23       public double getWeeklySalary()
24       {
25          return weeklySalary;
26       } // fim do método getWeeklySalary
27
28       // calcula os rendimentos; sobrescreve o método earnings em Employee
29       @Override
30       public double earnings()
31       {
32          return getWeeklySalary();
33       } // fim do método earnings
34
35       // retorna a representação String do objeto SalariedEmployee
36       @Override
37       public String toString()
38       {
39          return String.format( "salaried employee: %s\n%s: $%,.2f",
40             super.toString(), "weekly salary", getWeeklySalary() );
41       } // fim do método toString
42    } // fim da classe SalariedEmployee
```

Sobrescrever `earnings` torna esta classe concreta

Sobrescrever `toString` fornece uma representação `String` para esta classe

Figura 10.5 | A classe concreta `SalariedEmployee` estende a classe `abstract` `Employee`. (Parte 2 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5.3 Subclasse concreta HourlyEmployee

```
1  // Figura 10.6: HourlyEmployee.java
2  // Classe HourlyEmployee estende Employee.
3
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // salário por hora
7     private double hours; // horas trabalhadas durante a semana
8
9     // construtor de cinco argumentos
10    public HourlyEmployee( String first, String last, String ssn,
11       double hourlyWage, double hoursWorked )
12    {
13       super( first, last, ssn );
14       setWage( hourlyWage ); // valida a remuneração por hora
15       setHours( hoursWorked ); // valida as horas trabalhadas
16    } // fim do construtor HourlyEmployee com cinco argumentos
17
18    // configura a remuneração
19    public void setWage( double hourlyWage )
20    {
21       wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22    } // fim do método setWage
23
```

Figura 10.6 | Classe `HourlyEmployee` derivada de `Employee`. (Parte 1 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
24       // retorna a remuneração
25       public double getWage()
26       {
27          return wage;
28       } // fim do método getWage
29
30       // configura as horas trabalhadas
31       public void setHours( double hoursWorked )
32       {
33          hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34             hoursWorked : 0.0;
35       } // fim do método setHours
36
37       // retorna as horas trabalhadas
38       public double getHours()
39       {
40          return hours;
41       } // fim do método getHours
42
```

Figura 10.6 | Classe `HourlyEmployee` derivada de `Employee`. (Parte 2 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
43      // calcula os rendimentos; sobrescreve o método earnings em Employee
44      @Override
45      public double earnings()
46      {
47         if ( getHours() <= 40 ) // nenhuma hora extra
48            return getWage() * getHours();
49         else
50            return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
51      } // fim do método earnings
52
53      // retorna a representação de String do objeto HourlyEmployee
54      @Override
55      public String toString()
56      {
57      return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
58            super.toString(), "hourly wage", getWage(),
59            "hours worked", getHours() );
60      } // fim do método toString
61   } // fim da classe HourlyEmployee
```

Sobrescrever `earnings` torna esta classe concreta

Sobrescrever `toString` fornece uma representação `String` para esta classe

Figura 10.6 | Classe `HourlyEmployee` derivada de `Employee`. (Parte 3 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5.4 Subclasse concreta CommissionEmployee

```
1  // Figura 10.7: CommissionEmployee.java
2  // Classe CommissionEmployee estende Employee.
3
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // vendas brutas semanais
7     private double commissionRate; // porcentagem da comissão
8
9     // construtor de cinco argumentos
10    public CommissionEmployee( String first, String last, String ssn,
11       double sales, double rate )
12    {
13       super( first, last, ssn );
14       setGrossSales( sales );
15       setCommissionRate( rate );
16    } // fim do construtor CommissionEmployee de cinco argumentos
17
18    // configura a taxa de comissão
19    public void setCommissionRate( double rate )
20    {
21       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22    } // fim do método setCommissionRate
23
```

Figura 10.7 | Classe `CommissionEmployee` derivada de `Employee`. (Parte 1 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
24      // retorna a taxa de comissão
25      public double getCommissionRate()
26      {
27         return commissionRate;
28      } // fim do método getCommissionRate
29
30      // configura a quantidade de vendas brutas
31      public void setGrossSales( double sales )
32      {
33         grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34      } // fim do método setGrossSales
35
36      // retorna a quantidade de vendas brutas
37      public double getGrossSales()
38      {
39         return grossSales;
40      } // fim do método getGrossSales
41
42      // calcula os rendimentos; sobrescreve o método earnings em Employee
43      @Override
44      public double earnings()
45      {
46         return getCommissionRate() * getGrossSales();
47      } // fim do método earnings
```

Sobrescrever `earnings` torna esta classe concreta

Figura 10.7 | Classe `CommissionEmployee` derivada de `Employee`. (Parte 2 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
48
49      // retorna a representação String do objeto CommissionEmployee
50      @Override
51      public String toString()
52      {
53         return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
54            "commission employee", super.toString(),
55            "gross sales", getGrossSales(),
56            "commission rate", getCommissionRate() );
57      } // fim do método toString
58   } // fim da classe CommissionEmployee
```

Sobrescrever `toString` fornece uma representação `String` para esta classe

Figura 10.7 | Classe `CommissionEmployee` derivada de `Employee`. (Parte 3 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

COMO PROGRAMAR

8ª edição

10.5.5 Subclasse concreta indireta BasePlusCommissionEmployee

```
1  // Fig. 10.8: BaseplusCommissionEmployee.java
2  // Classe BasePlusCommissionEmployee estende a CommissionEmployee.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // salário de base por semana
7
8     // construtor de seis argumentos
9     public BasePlusCommissionEmployee( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       super( first, last, ssn, sales, rate );
13       setBaseSalary( salary ); // valida e armazena salário-base
14    } // fim do construtor BasePlusCommissionEmployee de seis argumentos
15
16    // configura o salário-base
17    public void setBaseSalary( double salary )
18    {
19       baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // não negativo
20    } // fim do método setBaseSalary
21
```

Fig. 10.8 | A classe `BasePlusCommissionEmployee` estende `CommissionEmployee`. (Parte 1 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
22      // retorna o salário-base
23      public double getBaseSalary()
24      {
25         return baseSalary;
26      } // fim do método getBaseSalary
27
28      // calcula os vencimentos; sobrescreve o método earnings em CommissionEmployee
29      @Override
30      public double earnings()
31      {
32         return getBaseSalary() + super.earnings();
33      } // fim do método earnings
34
35      // retorna a representação String do objeto BasePlusCommissionEmployee
36      @Override
37      public String toString()
38      {
39         return String.format( "%s %s; %s: $%,.2f",
40            "base-salaried", super.toString(),
41            "base salary", getBaseSalary() );
42      } // fim do método toString
43   } // fim da classe BasePlusCommissionEmployee
```

Se não sobrescrevermos `earning` nesta classe, herdamos a versão da superclasse `CommissionEmployee` e sua classe ainda é uma classe concreta

Sobrescrever `earnings` torna esta classe concreta

Fig. 10.8 | A classe `BasePlusCommissionEmployee` estende `CommissionEmployee`. (Parte 2 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5.6 Processamento polimórfico, operador instanceof e downcasting

- A Figura 10.9 cria um objeto de cada uma das quatro classes concretas.

  O programa manipula esses objetos não-polimorficamente, via as variáveis do próprio tipo do cada objeto e, então, polimorficamente, utilizando um array de variáveis `Employee`.
- Ao processar os objetos polimorficamente, o programa aumenta o salário-base de cada `BasePlusCommissionEmployee` por 10%

  Exige determinar o tipo de um objeto em tempo de execução.
- Por fim, o programa determina polimorficamente e gera a saída do tipo de cada objeto no array `Employee`. 18

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
1  // Figura 10.9: PayrollSystemTest.java
2  // Programa de teste da hierarquia Employee.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String[] args )
7     {
8        // cria objetos de subclasse
9        SalariedEmployee salariedEmployee =
10         new SalariedEmployee( “John”, “Smith”, “111-11-1111”, 800.00 );
11       HourlyEmployee hourlyEmployee =
12         new HourlyEmployee( “Karen”, “Price”, “222-22-2222”, 16.75, 40 );
13       CommissionEmployee commissionEmployee =
14         new CommissionEmployee(
15         “Sue”, “Jones”, “333-33-3333”, 10000, .06 );
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17         new BasePlusCommissionEmployee(
18         “Bob”, “Lewis”, “444-44-4444”, 5000, .04, 300 );
19
20       System.out.println( "Employees processed individually:\n" );
21
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23         salariedEmployee, "earned", salariedEmployee.earnings() );
```

Figura 10.9 | Programa de teste da hierarquia `Employee`. (Parte 1 de 6.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
24          System.out.printf( "%s\n%s: $%,.2f\n\n",
25            hourlyEmployee, "earned", hourlyEmployee.earnings() );
26          System.out.printf( "%s\n%s: $%,.2f\n\n",
27            commissionEmployee, "earned", commissionEmployee.earnings() );
28          System.out.printf( "%s\n%s: $%,.2f\n\n",
29            basePlusCommissionEmployee,
30            "earned", basePlusCommissionEmployee.earnings() );
31
32          // cria um array Employee de quatro elementos
33          Employee[] employees = new Employee[ 4 ];
34
35          // inicializa o array com Employees
36          employees[ 0 ] = salariedEmployee;
37          employees[ 1 ] = hourlyEmployee;
38          employees[ 2 ] = commissionEmployee;
39          employees[ 3 ] = basePlusCommissionEmployee;
40
41          System.out.println( "Employees processed polymorphically:\n" );
42
43          // processa genericamente cada elemento no employees
44          for ( Employee currentEmployee : employees )
45          {
46             System.out.println(currentEmployee ); // invoca toString
47
```

Não cria objetos `Employee` — apenas variáveis que podem referenciar objetos das subclasses `Employee`

Visa cada variável `Employee` em um objeto da subclasse `Employee`

Invoca `toString` polimorficamente

Figura 10.9 | Programa de teste da hierarquia `Employee`. (Parte 2 de 6.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
48            // determina se elemento é um BasePlusCommissionEmployee
49            if (currentEmployee instanceof BasePlusCommissionEmployee )
50            {
51               // downcast da referência de Employee para
52               // referência a BasePlusCommissionEmployee
53               BasePlusCommissionEmployee employee =
54                  ( BasePlusCommissionEmployee ) currentEmployee;
55
56               employee.setBaseSalary( 1.10 * employee.getBaseSalary() );
57
58               System.out.printf(
59                  "new base salary with 10%% increase is: $%,.2f\n",
60                  employee.getBaseSalary() );
61            } // fim do if
62
63            System.out.printf(
64               "earned $%,.2f\n\n", currentEmployee.earnings() );
65         } // for final
66
67         // obtém o nome do tipo de cada objeto no array employees
68         for ( int j = 0; j < employees.length; j++ )
69            System.out.printf( "Employee %d is a %s\n", j,
70               employees[ j ].getClass().getName() );
71      } // fim de main
72   } // fim da classe PayrollSystemTest
```

currentEmployee é um BasePlus-CommissionEmployee?

Esse downcast funciona porque currentEmployee é um BasePlus-CommissionEmployee

Invoca toString polimorficamente

Todo objeto em Java conhece seu próprio tipo

Figura 10.9 | Programa de teste da hierarquia `Employee`. (Parte 3 de 6.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

Figura 10.9 | Programa de teste da hierarquia `Employee`. (Parte 4 de 6.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00
```

Figura 10.9 | Programa de teste da hierarquia `Employee`. (Parte 5 de 6.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Figura 10.9 | Programa de teste da hierarquia `Employee`. (Parte 6 de 6.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Todas as chamadas ao método `toString` e `earnings` são resolvidas em tempo de execução, com base no tipo do objeto que `currentEmployee` referencia.
  Conhecido como **vinculação dinâmica** ou **vinculação tardia.**
  O Java decide qual método `toString` da classe é chamado em tempo de execução em vez de em tempo de compilação.
- Uma referência de superclasse pode ser utilizada para invocar somente métodos da superclasse — as implementações de método de subclasse são invocadas polimorficamente.
- Tentar invocar um método só de subclasse diretamente em uma referência de superclasse é um erro de compilação.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 10.3

Atribuir uma variável de superclasse a uma variável de subclasse (sem uma coerção explícita) é um erro de compilação.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 10.4

Se, em tempo de execução, a referência de um objeto de subclasse tiver sido atribuída a uma variável de uma das suas superclasses diretas ou indiretas, é aceitável fazer downcast da referência armazenada nessa variável de superclasse de volta a uma referência do tipo da subclasse. Antes de realizar essa coerção, utilize o operador `instanceof` *para assegurar que o objeto é de fato um objeto de um tipo de subclasse apropriado.*

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Erro comum de programação 10.4

Ao fazer o downcast de um objeto, ocorre uma `ClassCastException` *se em tempo de execução o objeto não tiver um relacionamento* é um *com o tipo especificado no operador de coerção. Só é possível fazer a coerção em uma referência no seu próprio tipo ou no tipo de uma das suas superclasses.*

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Cada objeto em Java conhece sua própria classe e pode acessar essas informações por meio do método **getClass**, que todas as classes herdam da classe **`Object`**. O método **`getClass`** retorna um objeto do tipo **Class** (do pacote **`java.lang`**), que contém as informações sobre o tipo do objeto, incluindo seu nome de classe. O resultado da chamada a `getClass` é utilizado para invocar **getName** a fim de obter o nome de classe do objeto.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 10.5

Embora o método real que é chamado dependa do tipo em tempo de execução do objeto que uma variável referencia, uma variável pode ser utilizada para invocar somente aqueles métodos que são membros do tipo dessa variável, o qual o compilador verifica.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.5.7 Resumo das atribuições permitidas entre variáveis de superclasse e de subclasse

- Há quatro modos de atribuição de referências de superclasse e subclasse a variáveis de tipos de subclasse e superclasse.
- Atribuir uma referência de superclasse a uma variável de superclasse é simples e direto.
- Atribuir uma referência de subclasse a uma variável de subclasse é simples e direto.
- Atribuir uma referência de subclasse a uma variável de superclasse é seguro, porque o objeto de subclasse *é um objeto de sua superclasse*.

  A variável da superclasse pode ser utilizada para referenciar apenas membros da superclasse.

  Se esse código referencia membros somente da subclasse por meio da variável de superclasse, o compilador informa erros.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Tentar atribuir uma referência de superclasse a uma variável de subclasse é um erro de compilação.

  Para evitar esse erro, a referência de superclasse deve sofrer uma coerção explícita para um tipo de subclasse.

  Em *tempo de execução, se o objeto referenciado não for um objeto de subclasse, ocorrerá uma exceção.*

  Use o operador `instanceof` para assegurar que tal coerção seja realizada somente se o objeto for um objeto de subclasse.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.6 Métodos e classes final

- Um método final em uma superclasse não pode ser sobrescrito em uma subclasse.
  Os métodos que são declarados `private` são implicitamente `final`, porque não é possível sobrescrevê-los em uma subclasse.
  Os métodos que são declarados `static` são implicitamente `final`.
  Uma declaração de método `final` nunca pode mudar, assim todas as subclasses utilizam a mesma implementação do método; e chamadas a métodos `final` são resolvidas em tempo de compilação — isso é conhecido como **vinculação estática.**

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Uma classe **final não pode ser uma superclasse (isto é, uma classe não pode estender uma classe** final*)*.

  Todos os métodos em uma classe `final` são implicitamente `final`.
- A classe `String` é um exemplo de uma classe `final`.

  Se fosse possível criar uma subclasse de `String`, os objetos dessa subclasse poderiam ser utilizados onde quer que `Strings` fossem esperadas.

  Como a classe `String` não pode ser estendida, os programas que utilizam `Strings` podem contar com a funcionalidade dos objetos `String` conforme especificada na Java API.

  Tornar a classe `final` também impede que programadores criem subclasses que poderiam driblar as restrições de segurança.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Observação de engenharia de software 10.5

Embora o método real que é chamado dependa do tipo em tempo de execução do objeto que uma variável referencia, uma variável pode ser utilizada para invocar somente aqueles métodos que são membros do tipo dessa variável, o qual o compilador verifica.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Observação de engenharia de software 10.6

Na Java API, a ampla maioria das classes não é declarada `final`. *Isso permite a herança e o polimorfismo. Entretanto, em alguns casos, é importante declarar classes* `final` *— em geral por questões de segurança.*

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7 Estudo de caso: criando e utilizando interfaces

- Nosso próximo exemplo reexamina o sistema de folha de pagamentos da Seção 10.5.
- Suponha que a empresa queria realizar várias operações contábeis em um aplicativo de contas a pagar único

  Calculando o lucro que deve ser pago a cada empregado

  Calcule o pagamento devido em cada uma de várias faturas (isto é, contas de mercadorias compradas)
- Ambas as operações têm a ver com a obtenção de algum tipo de quantia de pagamento.

  Para um funcionário, o pagamento se refere aos vencimentos do funcionário.

  Para uma fatura, o pagamento se refere ao custo total das mercadorias listadas na fatura.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- **Interfaces** oferece uma capacidade que exige que classes não relacionadas implementem um conjunto de métodos comuns.
- Interfaces definem e padronizam como coisas, pessoas e sistemas, podem interagir entre si.

  Exemplo: Os controles em um rádio servem como uma interface entre os usuários do rádio e os componentes internos do rádio.

  É possível realizar somente um conjunto limitado de operações (por exemplo, alterar a estação, ajustar o volume, escolher entre AM e FM)

  Diferentes rádios podem implementar os controles de diferentes maneiras (por exemplo, uso de botões, sintonizadores, comandos de voz).

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- A interface especifica *quais* operações um rádio devem permitir que os usuários realizem, mas não especifica *como* essas operações são realizadas.
- Uma interface Java descreve o conjunto de métodos que pode ser chamado em um objeto.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Uma **declaração de interface** inicia com a palavra-chave **interface** e contém somente constantes e métodos `abstract`.

  Todos os membros de interface devem ser `public`.

  Interfaces não podem especificar nenhum detalhe da implementação, como declarações de métodos concretos e variáveis de instância.

  Todos os métodos declarados em uma interface são implicitamente métodos `public abstract`

  Todos os campos são implicitamente `public`, `static` e `final`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 10.1

De acordo com o Capítulo 9 da Java Language Specification, é estilisticamente correto declarar os métodos de uma interface sem as palavras-chave `public` *e* `abstract`*, porque elas são redundantes nas declarações de método de interface. De modo semelhante, as constantes devem ser declaradas sem as palavras-chave* `public`*,* `static` *e* `final`*, porque elas também são redundantes.*

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Para utilizar uma interface, uma classe concreta deve especificar que ela **implementa** a interface e deve declarar cada método na interface com a assinatura especificada.

  Adicione a palavra-chave `implements` e o nome da interface no fim da primeira linha da declaração da classe.
- Uma classe que não implementa todos os métodos da interface é uma classe abstrata e deve ser declarada `abstract`.
- Implementar uma interface é como assinar um contrato com o compilador que afirma, "Irei declarar todos os métodos especificados pela interface ou irei declarar minha classe `abstract`."

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 10.6

Falhar em implementar qualquer método de uma interface em uma classe concreta que `implementa` *a interface resulta em um erro de compilação indicando que a classe deve ser declarada* `abstract`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Uma interface é geralmente utilizada quando classes díspares (isto é, não-relacionadas) precisam compartilhar métodos e constantes comuns.
  Permite que objetos de classes não relacionadas sejam processados polimorficamente.
  Você pode criar uma interface que descreve a funcionalidade desejada e então implementar essa interface em quaisquer classes que requerem essa funcionalidade.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Uma interface costuma ser utilizada no lugar de uma classe `abstract` quando não há nenhuma implementação padrão a herdar — isto é, nenhum campo e nenhuma implementação padrão de método.
- Assim como as classes `public abstract`, as interfaces são geralmente tipos `public`.
- Uma interface `public` deve ser declarada em um arquivo com o mesmo nome da interface e a extensão de arquivo `.java`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.1 Desenvolvendo uma hierarquia Payable

- O seguinte exemplo constrói um aplicativo que pode determinar pagamentos para empregados e faturas igualmente.
  As classes `Invoice` e `Employee` representam aspectos para os quais a empresa deve ser capaz de calcular um valor de pagamento.
  Ambas as classes implementam a interface `Payable`, desse modo, um programa pode invocar o método `getPaymentAmount` em objetos `Invoice` e, igualmente, em objetos `Employee`.
  Permite o processamento polimórfico de `Invoice`s e `Employee`s.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Boa prática de programação 10.2

Ao declarar um método em uma interface, escolha um nome de método que descreva o propósito do método de uma maneira geral, pois o método pode ser implementado por muitas classes não relacionadas.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- A Figura 10.10 mostra a hierarquia de contas a pagar.
- A UML separa uma interface de outras classes colocando «interface» acima do nome da interface.
- A UML expressa o relacionamento entre uma classe e uma interface por meio de um relacionamento conhecido como **realização.**

  Dizemos que uma classe "realiza", ou implementa, os métodos de uma interface.

  Um diagrama de classe modela uma realização como uma seta tracejada com uma ponta oca apontando da implementação da classe para a interface.
- Uma subclasse herda relacionamentos de realização da sua superclasse.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

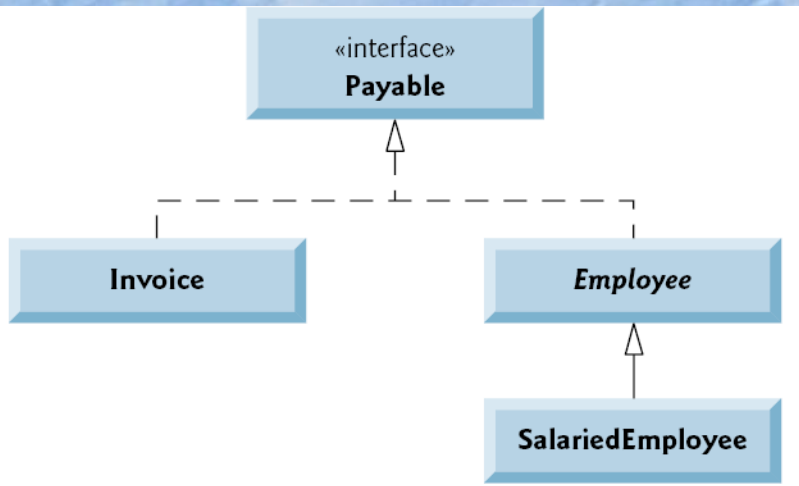

Figura 10.10 | Diagrama de classe em UML da hierarquia da interface `Payable`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.2 Interface Payable

- A Figura 10.11 mostra a declaração da interface `Payable`.
- Métodos de interface sempre são `public` e `abstract`, de modo que não seja necessário declará-los como tais.
- Interfaces podem ter qualquer número de métodos
- Observe que as interfaces também podem conter campos que são implicitamente `final` e `static`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
1  // Figura 10.11: Payable.java
2  // Declaração da interface Payable.
3
4  public interface Payable
5  {
6     double getPaymentAmount(); // calcula pagamento; nenhuma implementação
7  } // fim da interface Payable
```

Figura 10.11 | Declaração da interface `payable`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.3 Classe Invoice

- O Java não permite que subclasses estendam mais de uma superclasse, mas permite que uma classe estenda uma superclasse e implemente quantas interfaces ela precisar.
- Para implementar mais de uma interface, utilize uma lista separada por vírgulas de nomes de interfaces depois da palavra-chave `implements` na declaração de classe, como em:

```
public class NomeDaClasse extends NomeDaSuperclasse
   implements PrimeiraInterface, SegundaInterface, ...
```

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
1  // Figura 10.12: Invoice.java
2  // Classe Invoice que implementa Payable.
3
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // construtor com quatro argumentos
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
16       partDescription = description;
17       setQuantity( count ); // valida e armazena a quantidade
18       setPricePerItem( price ); // valida e armazena o preço por item
19    } // fim do construtor Invoice de quatro argumentos
20
```

A classe estende `Object` (implicitamente) e implementa a interface `Payable`

Figura 10.12 | A classe `Invoice` que implementa `Payable`. (Parte 1 de 4.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
21      // configura número de peças
22      public void setPartNumber( String part )
23      {
24         partNumber = part; // deve validar
25      } // fim do método setPartNumber
26
27      // obtém o número da peça
28      public String getPartNumber()
29      {
30         return partNumber;
31      } // fim do método getPartNumber
32
33      // configura a descrição
34      public void setPartDescription( String description )
35      {
36         partDescription = description; // deve validar
37      } // fim do método setPartDescription
38
39      // obtém a descrição
40      public String getPartDescription()
41      {
42         return partDescription;
43      } // fim do método getPartDescription
44
```

Figura 10.12 | A classe `Invoice` que implementa `Payable`. (Parte 2 de 4.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
45      // configura a quantidade
46      public void setQuantity( int count )
47      {
48         quantity = ( count < 0 ) ? 0 : count; // a quantidade não pode ser negativa
49      } // fim do método setQuantity
50
51      // obtém a quantidade
52      public int getQuantity()
53      {
54         return quantity;
55      } // fim do método getQuantity
56
57      // configura preço por item
58      public void setPricePerItem( double price )
59      {
60         pricePerItem = ( price < 0.0 ) ? 0.0 : price; // valida o preço
61      } // fim do método setPricePerItem
62
63      // obtém preço por item
64      public double getPricePerItem()
65      {
66         return pricePerItem;
67      } // fim do método getPricePerItem
68
```

Figura 10.12 | A classe `Invoice` que implementa `Payable`. (Parte 3 de 4.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
69       // retorno da representação de String do objeto Invoice
70       @Override
71       public String toString()
72       {
73          return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
74          "invoice", "part number", getPartNumber(), getPartDescription(),
75          "quantity", getQuantity(), "price per item", getPricePerItem() );
76       } // fim do método toString
77
78       // método requerido para executar o contrato com a interface Payable
79       @Override
80       public double getPaymentAmount()
81       {
82          return getQuantity() * getPricePerItem(); // calcula custo total
83       } // fim do método getPaymentAmount
84    } // fim da classe Invoice
```

Fornecer uma implementação do(s) método(s) da interface torna esta classe concreta

Figura 10.12 | A classe `Invoice` que implementa `Payable`. (Parte 4 de 4.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Observação de engenharia de software 10.7

Todos os objetos de uma classe que implementam múltiplas interfaces têm o relacionamento do tipo "é um" com cada tipo de interface implementado.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.4 Modificando a classe Employee para implementar a interface Payable

- Quando uma classe implementa uma interface, ela faz um contrato com o compilador.
  A classe implementará cada um dos métodos na interface ou essa classe será declarada `abstract`.
  Se a última opção for escolhida, não precisamos declarar os métodos de interface como `abstract` na classe `abstract` — eles já são implicitamente declarados como tais na interface.
  Qualquer subclasse concreta da classe `abstract` deve implementar os métodos de interface para cumprir o contrato.
  Se a subclasse não fizer isso, ela também deverá ser declarada `abstract`.
- Cada subclasse `Employee` direta herda o contrato da superclasse para implementar o método `getPaymentAmount` e assim deve implementar esse método para tornar-se uma classe concreta na qual os objetos podem ser instanciados.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
1  // Figura 10.13: Employee.java
2  // Superclasse abstrata Employee implementa Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // construtor com três argumentos
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // fim do construtor Employee com três argumentos
17
18    // configura o nome
19    public void setFirstName( String first )
20    {
21       firstName = first; // deve validar
22    } // fim do método setFirstName
23
```

Classe abstrata estende `Object` (implicitamente) e implementa a interface `Payable`

Figura 10.13 | A classe `Employee` que implementa `Payable`. (Parte 1 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
24       // retorna o nome
25       public String getFirstName()
26       {
27          return firstName;
28       } // fim do método getFirstName
29
30       // configura o sobrenome
31       public void setLastName( String last )
32       {
33          lastName = last; // deve validar
34       } // fim do método setLastName
35
36       // retorna o sobrenome
37       public String getLastName()
38       {
39          return lastName;
40       } // fim do método getLastName
41
42       // configura o CIC
43       public void setSocialSecurityNumber( String ssn )
44       {
45          socialSecurityNumber = ssn; // deve validar
46       } // fim do método setSocialSecurityNumber
47
```

Figura 10.13 | A classe `Employee` que implementa `Payable`. (Parte 2 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
48       // retorna o CIC
49       public String getSocialSecurityNumber()
50       {
51          return socialSecurityNumber;
52       } // fim do método getSocialSecurityNumber
53
54       // retorna a representação de String do objeto Employee
55       @Override
56       public String toString()
57       {
58          return String.format( "%s %s\nsocial security number: %s",
59             getFirstName(), getLastName(), getSocialSecurityNumber() );
60       } // fim do método toString
61
62       // Nota: não implementamos o método Payable getPaymentAmount aqui, mas essa
63       // classe deve ser declarada abstrata para evitar um erro de compilação.
64    } // fim da classe abstrata Employee
```

Não implementamos o método da interface; portanto, essa classe deve ser declarada `abstract`

Figura 10.13 | A classe `Employee` que implementa `Payable`. (Parte 3 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.5 Modificando a classe SalariedEmployee para uso na hierarquia Payable

- A Figura 10.14 contém uma classe `SalariedEmployee` modificada que estende `Employee` e cumpre o contrato da superclasse `Employee` para implementar o método `Payable` `getPaymentAmount`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
1  // Figura 10.14: SalariedEmployee.java
2  // Classe SalariedEmployee estende Employee que implementa Payable.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // construtor com quatro argumentos
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // passa para o construtor Employee
13       setWeeklySalary( salary ); // valida e armazena o salário
14    } // fim do construtor SalariedEmployee com quatro argumentos
15
16    // configura o salário
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // fim do método setWeeklySalary
21
```

Figura 10.14 | A classe `SalariedEmployee` que implementa o método `getPaymentAmount` da interface `Payable`. (Parte 1 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
22      // retorna o salário
23      public double getWeeklySalary()
24      {
25         return weeklySalary;
26      } // fim do método getWeeklySalary
27
28      // calcula vencimentos; implementa o método Payable da
29      // interface que era abstrata na superclasse Employee
30      @Override
31      public double getPaymentAmount()
32      {
33         return getWeeklySalary();
34      } // fim do método getPaymentAmount
35
36      // retorna a representação String do objeto SalariedEmployee
37      @Override
38      public String toString()
39      {
40         return String.format( "salaried employee: %s\n%s: $%,.2f",
41            super.toString(), "weekly salary", getWeeklySalary() );
42      } // fim do método toString
43   } // fim da classe SalariedEmployee
```

Fornecer uma implementação do(s) método(s) da interface torna esta classe concreta

Figura 10.14 | A classe `SalariedEmployee` que implementa o método `getPaymentAmount` da interface `Payable`. (Parte 2 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- Um objeto de quaisquer subclasses de uma classe que `implementa` uma interface também pode ser pensado como um objeto do tipo interface.
- Portanto, assim como podemos atribuir a referência de um objeto `SalariedEmployee` a uma variável `Employee` da superclasse, podemos atribuir a referência de um objeto `SalariedEmployee` a uma variável `Payable` da interface.
- `Invoice` implementa `Payable`, portanto um objeto `Invoice` também *é um* objeto *`Payable`*, *e podemos atribuir a referência de um objeto `Invoice` a uma variável `Payable`*.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Observação de engenharia de software 10.8

Quando um parâmetro de método é declarado com uma superclasse ou tipo de interface, o método processa o objeto recebido polimorficamente como um argumento.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 10.9

Utilizando uma referência de superclasse, podemos invocar polimorficamente qualquer método declarado na superclasse e suas superclasses (por exemplo, a classe `Object`*). Utilizando uma referência de interface, podemos invocar polimorficamente qualquer método declarado na interface, suas superinterfaces (uma interface pode estender outra) e na classe* `Object` *— uma variável de tipo de interface deve referenciar um objeto para chamar métodos, e todos os objetos têm os métodos da classe* `Object`*.*

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.6 Utilizando a interface Payable para processar Invoices e Employees polimorficamente

```
1  // Figura 10.15: PayableInterfaceTest.java
2  // Testa a interface Payable.
3
4  public class PayableInterfaceTest
5  {
6     public static void main( String[] args )
7     {
8        // cria array Payable de quatro elementos
9        Payable[] payableObjects = new Payable[ 4 ];
10
11       // preenche o array com objetos que implementam Payable
12       payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13       payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14       payableObjects[ 2 ] =
15          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16       payableObjects[ 3 ] =
17          new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19       System.out.println(
20          "Invoices and Employees processed polymorphically:\n" );
21
```

Cria um array de quatro variáveis `Payable`

Visa cada variável `Payable` no objeto de uma classe que implementa a interface `Payable`

Figura 10.15 | Programa de teste da interface `Payable` que processa `Invoices` e `Employees` polimorficamente. (Parte 1 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.


```
22          // processa genericamente cada elemento no array payableObjects
23          for ( Payable currentPayable : payableObjects )
24          {
25             // gera saída de currentPayable e o pagamento apropriado
26             System.out.printf( "%s \n%s: $%,.2f\n\n",
27                currentPayable.toString(),
28                "payment due", currentPayable.getPaymentAmount() );
29          } // for final
30       } // fim de main
31    } // fim da classe PayableInterfaceTest
```

Figura 10.15 | Programa de teste da interface `Payable` que processa `Invoices` e `Employees` polimorficamente. (Parte 2 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Figura 10.15 | Programa de teste da interface `Payable` que processa `Invoices` e `Employees` polimorficamente. (Parte 3 de 3.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.7.7 Interfaces comuns da Java API

- As interfaces da Java API permitem que você utilize suas próprias classes dentro das estruturas fornecidas pelo Java, como comparar os objetos dos seus próprios tipos e criar tarefas que podem executar concorrentemente com outras tarefas no mesmo programa.
- A Figura 10.16 apresenta uma breve visão geral de algumas das interfaces mais populares da Java API que utilizamos em *Java, como programar, oitava edição.*

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

Interface	Descrição
Interfaces listener de eventos com GUIs	Você utiliza interfaces gráficas com o usuário (GUIs) todos os dias. No navegador Web, você poderia digitar o endereço de um site Web a visitar, ou clicar em um botão para voltar a um site anterior. O navegador responde à sua interação e realiza a tarefa desejada. Sua interação é conhecida como um evento e o código que o navegador utiliza para responder a um evento é conhecido como um handler de eventos. No Capítulo 14, Componentes GUI: Parte 1, e no Capítulo 25, Componentes GUI: Parte 2, você aprenderá a construir GUI e rotinas de tratamento de evento que respondem a interações de usuário. Handlers de eventos são declarados em classes que implementam uma interface ouvinte de eventos apropriada. Cada interface ouvinte de eventos especifica um ou mais métodos que devem ser implementados para responder a interações de usuário.
`SwingConstants`	Contém um conjunto de constantes utilizado em programação de GUI para posicionar elementos da GUI na tela. Exploramos a programação de GUI nos capítulos 14 e 25.

Figura 10.16 | Interfaces comuns da Java API. (Parte 2 de 2.)

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

10.8 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando com polimorfismo

- Classes de formas têm muitas semelhanças.
- Com a herança, podemos "fatorar" os recursos comuns de todas as três classes e colocá-los em uma única superclasse de forma.
- Então, utilizando variáveis do tipo de superclasse, podemos manipular polimorficamente os objetos de todos os três tipos de forma.
- Remover a redundância no código resultará em um programa menor, mais flexível e mais fácil de manter.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

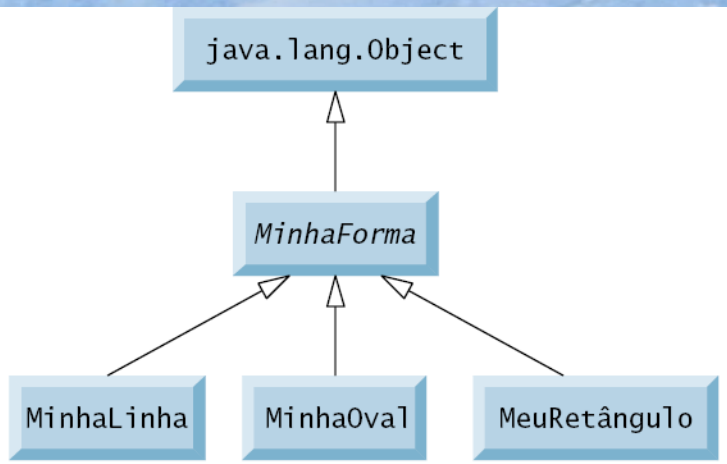

Figura 10.17 | Hierarquia `MinhaForma`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

- A classe `MyBoundedShape` pode ser usada para fatorar as características comuns das classes `MyOval` e `MyRectangle`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.

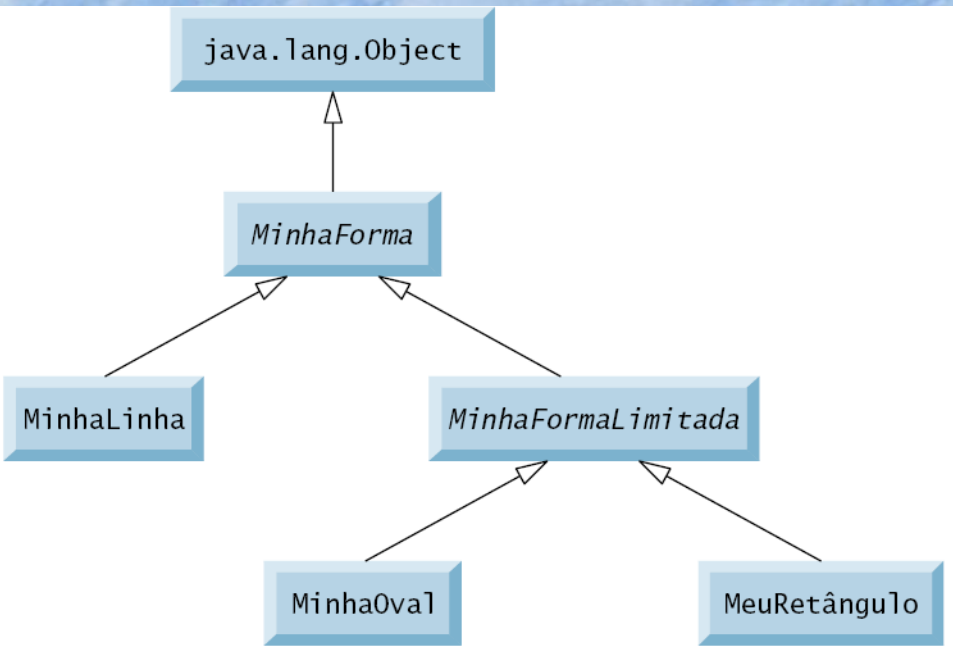

Figura 10.18 | Hierarquia `MinhaForma` com `MinhaFormaLimitada`.

(C) 2010 Pearson Education, Inc. Todos os direitos reservados.