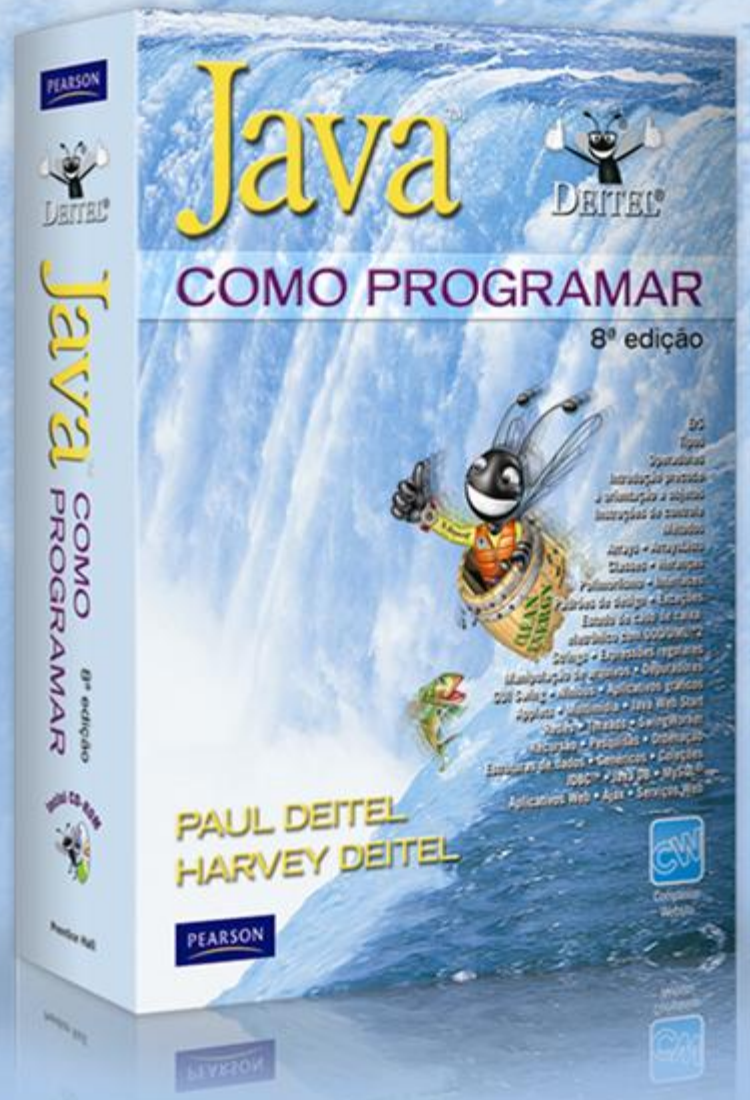
Capítulo 9
Programação orientada a objetos: herança



Java™ Como Programar, 8/E

OBJETIVOS

Neste capítulo, você aprenderá:

- Como a herança promove a capacidade de reutilização de software.
- As noções de superclasses e subclasses.
- A utilizar a palavra-chave `extends` para criar uma classe que herda atributos e comportamentos de outra classe.
- A utilizar o modificador de acesso `protected` para fornecer acesso de métodos de subclasse a membros de superclasse.
- A acessar membros de superclasse com `super`.
- Como os construtores são utilizados em hierarquias de herança.
- Os métodos da classe `Object`, a superclasse direta ou indireta de todas as classes em Java.

9.1 Introdução

- **Herança**

  Uma forma de reutilização de software em que uma nova classe é criada absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas.

  Permite economizar tempo durante o desenvolvimento de um programa baseando novas classes no software existente testado, depurado e de alta qualidade.

  Aumenta a probabilidade de que um sistema será implementado e mantido eficientemente.

- Ao criar uma classe, em vez de declarar membros completamente novos, você pode designar que a nova classe deve herdar membros de uma classe existente.

  Classe existente na **superclasse.**

  Nova classe é a **subclasse.**
- Cada subclasse pode ser uma superclasse de futuras subclasses.
- Uma subclasse pode adicionar seus próprios campos e métodos.
- Uma subclasse é mais específica que sua superclasse e representa um grupo mais especializado de objetos.
- A subclasse expõe os comportamentos da sua superclasse e pode adicionar comportamentos que são específicos à subclasse.

  É por isso que a herança é às vezes conhecida como **especialização.**

- A **superclasse direta** é a superclasse a partir da qual a subclasse herda explicitamente.
- Uma **superclasse indireta** é qualquer classe acima da superclasse direta na **hierarquia de classes**.
- A hierarquia de classes inicia com a classe `Object` (no pacote `java.lang`)

  Toda classe em Java **estende** (ou "herda de") `Object` direta ou indiretamente.
- O Java só suporta **herança simples**, na qual cada classe é derivada de exatamente uma superclasse direta.

- Distinguimos entre o **relacionamento *é um*** e o **relacionamento tem um.**
- *É um* representa a herança.

  Em um relacionamento *é um*, um objeto de uma subclasse também pode ser tratado como um objeto de sua superclasse.
- *Tem um* representa composição.

  Em um relacionamento *tem um*, um objeto contém como membros referências a outros objetos.

9.2 Superclasses e subclasses

- A Figura 9.1 lista vários exemplos simples de superclasses e subclasses.

  As superclasses tendem a ser "mais gerais" e as subclasses "mais específicas".
- Como cada objeto de subclasse *é um* objeto de sua superclasse e uma superclasse pode ter muitas subclasses, o conjunto de objetos representado por uma superclasse é, em geral, maior que o conjunto de objetos representado por qualquer uma de suas subclasses.

Superclasse	Subclasses
Aluno	AlunoDeGraduação, AlunoDePósGraduação
Forma	Círculo, Triângulo, Retângulo, Esfera, Cubo
Financiamento	FinanciamentoDeCarro, FinanciamentoDeCasa
Empregado	CorpoDocente, Funcionário
ContaBancária	ContaCorrente, ContaPoupança

Figura 9.1 | Exemplos de herança.

- Uma superclasse existe em um relacionamento hierárquico com suas subclasses.
- A Figura 9.2 mostra um exemplo de hierarquia de classes na comunidade universitária. Também chamado de **hierarquia de herança.**
- Cada seta na hierarquia representa um relacionamento *é um.*
- Siga as setas para cima na hierarquia de classes

  "um `Funcionário` *é um* `MembroDaComunidade`"

  "um `Professor` *é um membro do* `CorpoDocente`."
- `MembroDaComunidade` é a superclasse direta de `Funcionários`, `AlunoDeGraduação` e `AlunoDePósGraduação` e é uma superclasse indireta de todas as outras classes no diagrama.
- A partir da parte inferior, você pode seguir as setas e aplicar um relacionamento *é um* até a superclasse na parte superior.

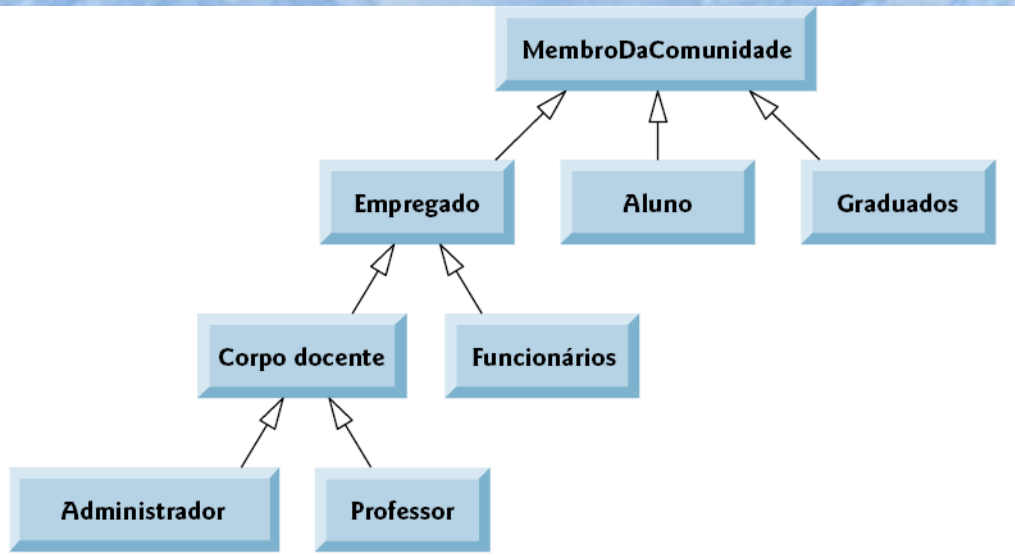


Figura 9.2 | Hierarquia de herança `MembrosDaComunidade` da universidade.

- A Figura 9.3 mostra uma hierarquia de herança de `Forma`.
- É possível seguir as setas desde a parte inferior do diagrama até a superclasse na parte superior nessa hierarquia de classes para identificar vários relacionamentos *é um*.

  Um `Triângulo` *é uma* `FormaBiDimensional` e *é uma* `Forma`

  Uma `Esfera` *é uma* `FormaTriDimensional` e *é uma* `Forma`.

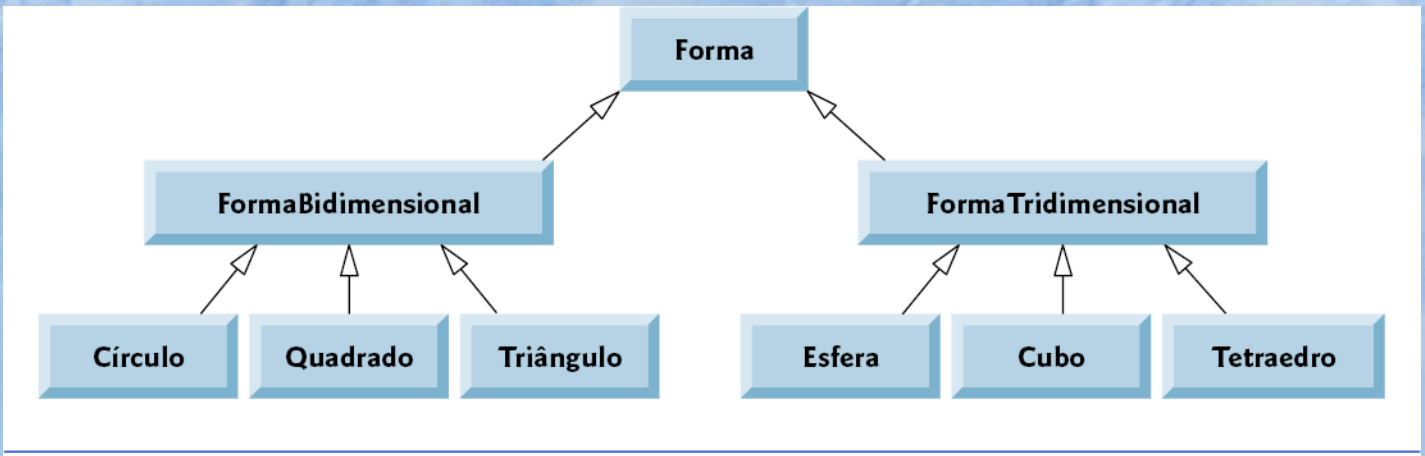


Figura 9.3 | Hierarquia de herança para Formas.

- Nem todo relacionamento de classe é um relacionamento de herança.
- *tem um*

  Criam classes compondo classes existentes.

  Exemplo: Dadas as classes `Empregado`, `DataDeNascimento` e `NúmeroDeTelefone`, é incorreto dizer que `Empregado` *é um* `DataDeNascimento` o que `Empregado` *é um* `NúmeroDeTelefone`. Mas um `Empregado` *tem uma* `DataDeNascimento`, e um `Empregado` *tem um* `NúmeroDeTelefone`.

- Os objetos de todas as classes que estendem uma superclasse comum podem ser todos tratados como objetos dessa superclasse.

  Seus aspectos comuns são expressos nos membros da superclasse.

- Problemas de herança

  Uma subclasse pode herdar métodos que ela não necessita ou que não deveria ter.

  Mesmo quando um método de superclasse é adequado a uma subclasse, essa subclasse precisa frequentemente de uma versão personalizada do método.

  A subclasse pode **sobrescrever** (redefinir) o método de superclasse com uma implementação apropriada.

9.3 Membros protected

- Os membros `public` de uma classe são acessíveis onde quer que o programa tenha uma referência a um objeto dessa classe ou uma de suas subclasses.
- Os membros `private` de uma classe são acessíveis apendas dentro da própria classe.
- Utilizar acesso **protected** oferece um nível intermediário de acesso entre `public` e `private`.
- Os membros `protected` de uma superclasse podem ser acessados por membros dessa superclasse, por membros de suas subclasses e por membros de outras classes no mesmo pacote.
- Os membros `protected` também têm acesso de pacote.
- Todos os membros `public` e `protected` de uma superclasse retêm seu modificador de acesso original quando se tornam membros da subclasse.

- Os membros `private` de uma superclasse permanecem ocultos em suas subclasses.

  Eles somente podem ser acessados pelos métodos `public` ou `protected` herdados da superclasse.
- Os métodos de subclasse podem referenciar membros `public` e `protected` herdados da superclasse simplesmente utilizando os nomes de membro.
- Quando um método de subclasse sobrescrever um método de superclasse herdado, o método de superclasse pode ser acessado a partir da subclasse precedendo o nome do método de superclasse com a palavra-chave **super** e um ponto (`.`) separador.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.1

Os métodos de uma subclasse não acessam membros `private` *diretamente de sua superclasse. Uma subclasse pode alterar o estado de variáveis de instância* `private` *da superclasse somente por meio de métodos não* `private` *fornecidos na superclasse e herdados pela subclasse.*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.2

Declarar variáveis de instância `private` *ajuda-lhe a testar, depurar e modificar sistemas corretamente. Se uma subclasse pudesse acessar variáveis de instância* `private` *da sua superclasse, classes que herdam dessa subclasse também poderiam acessar as variáveis de instância. Isso propagaria acesso ao que devem ser variáveis de instância* `private` *e os benefícios do ocultamento de informações seriam perdidos.*

9.4 Relacionamento entre superclasses e subclasses

- Hierarquia de herança que contém tipos de empregados no aplicativo de folha de pagamentos de uma empresa.
- Os empregados comissionados recebem uma porcentagem das suas vendas.
- Funcionários assalariados-comissionados recebem um salário-base mais uma porcentagem sobre suas vendas.

9.4.1 Criando e utilizando uma classe CommissionEmployee

- A classe `CommissionEmployee` (Figura 9.4) **estende** a classe **Object** (do pacote `java.lang`).

  `CommissionEmployee` herda métodos de `Object`.

  Se você não especificar explicitamente qual classe uma nova classe estende, a classe estenderá `Object` implicitamente.


```
1  // Figura 9.4: CommissionEmployee.java
2  // Classe CommissionEmployee representa um funcionário
3  // que recebeu uma porcentagem das vendas brutas.
4  public class CommissionEmployee extends Object
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // vendas brutas semanais
10    private double commissionRate; // porcentagem da comissão
11
12    // construtor de cinco argumentos
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // chamada implícita para o construtor Object ocorre aqui
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // valida e armazena as vendas brutas
21       setCommissionRate( rate ); // valida e armazena a taxa de comissão
22    } // fim do construtor CommissionEmployee de cinco argumentos
```

`extends Object` não é exigido; isso será feito implicitamente

Figura 9.4 | A classe `CommissionEmployee` representa um empregado pago com uma porcentagem das vendas brutas. (Parte 1 de 5.)

```
23
24       // configura o nome
25       public void setFirstName( String first )
26       {
27          firstName = first; // deve validar
28       } // fim do método setFirstName
29
30       // retorna o nome
31       public String getFirstName()
32       {
33          return firstName;
34       } // fim do método getFirstName
35
36       // configura o sobrenome
37       public void setLastName( String last )
38       {
39          lastName = last; // deve validar
40       } // fim do método setLastName
41
```

Figura 9.4 | A classe `CommissionEmployee` representa um empregado pago com uma porcentagem das vendas brutas. (Parte 2 de 5.)

```
42       // retorna o sobrenome
43       public String getLastName()
44       {
45          return lastName;
46       } // fim do método getLastName
47
48       // configura o CIC
49       public void setSocialSecurityNumber( String ssn )
50       {
51          socialSecurityNumber = ssn; // deve validar
52       } // fim do método setSocialSecurityNumber
53
54       // retorna o CIC
55       public String getSocialSecurityNumber()
56       {
57          return socialSecurityNumber;
58       } // fim do método getSocialSecurityNumber
59
60       // configura a quantidade de vendas brutas
61       public void setGrossSales( double sales )
62       {
63          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64       } // fim do método setGrossSales
```

Figura 9.4 | A classe `CommissionEmployee` representa um empregado pago com uma porcentagem das vendas brutas. (Parte 3 de 5.)


```
65
66       // retorna a quantidade de vendas brutas
67       public double getGrossSales()
68       {
69          return grossSales;
70       } // fim do método getGrossSales
71
72       // configura a taxa de comissão
73       public void setCommissionRate( double rate )
74       {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76       } // fim do método setCommissionRate
77
78       // retorna a taxa de comissão
79       public double getCommissionRate()
80       {
81          return commissionRate;
82       } // fim do método getCommissionRate
83
```

Figura 9.4 | A classe `CommissionEmployee` representa um empregado pago com uma porcentagem das vendas brutas. (Parte 4 de 5.)

```
84      // calcula os lucros
85      public double earnings()
86      {
87         return commissionRate * grossSales;
88      } // fim do método earnings
89
90      // retorna a representação String do objeto CommissionEmployee
91      @Override // indica que esse método sobrescreve um método de superclasse
92      public String toString()
93      {
94         return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
95            "commission employee", firstName, lastName,
96            "social security number", socialSecurityNumber,
97            "gross sales", grossSales,
98            "commission rate", commissionRate );
99      } // fim do método toString
100  } // fim da classe CommissionEmployee
```

`toString` sobrescrito personaliza o modo como o método funciona para `CommissionEmploye`; `@Override` ajuda o compilador a assegurar que o método tem a mesma assinatura que o método na superclasse

Figura 9.4 | A classe `CommissionEmployee` representa um empregado pago com uma porcentagem das vendas brutas. (Parte 5 de 5.)

- Construtores não são herdados.
- A primeira tarefa de um construtor de subclasse é chamar o construtor da sua superclasse direta, explícita ou implicitamente.

  Assegura que as variáveis de instância herdadas da superclasse sejam inicializadas adequadamente.
- Se o código não incluir uma chamada explícita para o construtor de superclasse, o Java chama implicitamente o construtor padrão (ou construtor sem argumento) da superclasse.
- O construtor padrão de uma classe chama o construtor padrão ou sem argumentos da superclasse.

- `toString` é um dos métodos que toda classe herda direta ou indiretamente da classe `Object`.

  Retorna uma `String` representando um objeto.

  Chamado implicitamente sempre que um objeto deve ser convertido em uma representação `String`.
- O método `toString` da classe `Object` retorna uma `String` que inclui o nome da classe do objeto.

  Isso é principalmente um espaço reservado que pode ser sobrescrito por uma subclasse para especificar uma representação `String` adequada.

- Para sobrescrever um método de superclasse, uma subclasse deve declarar um método com a mesma assinatura que o método de superclasse
- **Anotação @Override**

  Indica que um método deve sobrescrever um método de superclasse com a mesma assinatura.

  Se ele não fizer isso, ocorre um erro de compilação.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 9.1

Utilizar uma assinatura de método incorreta ao tentar sobrescrever um método de superclasse resulta em uma sobrecarga de método não intencional que pode levar a erros de lógica sutis.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 9.1

Declare métodos sobrescritos com a notação `@Override` *para assegurar em tempo de compilação que você definiu as assinaturas corretamente. Sempre é melhor localizar erros em tempo de compilação em vez de em tempo de execução.*

Erro comum de programação 9.2

É um erro de sintaxe sobrescrever um método com um modificador de acesso mais restrito — um método `public` *da superclasse não pode tornar-se um método* `protected` *ou* `private` *na subclasse; um método* `protected` *da superclasse não pode tornar-se um método* `private` *na subclasse. Fazer isso quebraria o relacionamento* é um *em que se exige que todos os objetos de subclasse sejam capazes de responder a chamadas de método que são feitas para os métodos* `public` *declarados na superclasse. Se um método* `public`*, por exemplo, pudesse ser sobrescrito como um método* `protected` *ou* `private`*, os objetos de subclasse não seriam capazes de responder às mesmas chamadas de método como objetos de superclasse. Uma vez que um método é declarado* `public` *em uma superclasse, o método permanece* `public` *para todas as subclasses diretas e indiretas da classe.*


```
1  // Figura 9.5: CommissionEmployeeTest.java
2  // Programa de teste da classe CommissionEmployee.
3
4  public class CommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instancia o objeto CommissionEmployee
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // obtém os dados de empregado comissionado
13       System.out.println(
14          "Employee information obtained by get methods: \n" );
15       System.out.printf( "%s %s\n", "First name is",
16          employee.getFirstName() );
17       System.out.printf( "%s %s\n", "Last name is",
18          employee.getLastName() );
19       System.out.printf( "%s %s\n", "Social security number is",
20          employee.getSocialSecurityNumber() );
21       System.out.printf( "%s %.2f\n", "Gross sales is",
22          employee.getGrossSales() );
23       System.out.printf( "%s %.2f\n", "Commission rate is",
24          employee.getCommissionRate() );
```

Figura 9.5 | Programa de teste da classe `CommissionEmployee`. (Parte 1 de 2.)


```
25
26          employee.setGrossSales( 500 ); // configura vendas brutas
27          employee.setCommissionRate( .1 ); // configura a taxa de comissão
28
29          System.out.printf( "\n%s:\n\n%s\n",
30             "Updated employee information obtained by toString", employee );
31       } // fim de main
32    } // fim da classe CommissionEmployeeTest
```

A chamada implícita ao método `toString` ocorre aqui

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Figura 9.5 | Programa de teste da classe `CommissionEmployee`. (Parte 2 de 2.)

9.4.2 Criando e utilizando uma classe BasePlus-CommissionEmployee

- A classe `BasePlusCommissionEmployee` (Figura 9.6) contém um nome, sobrenome, número de seguro social, quantidade de vendas brutas, taxa de comissão e salário-base.

  Tudo exceto salário-base é em comum com a classe `CommissionEmployee`.
- Os serviços `public` da classe `BasePlusCommissionEmployee` incluem um construtor e os métodos `earnings`, `toString` e *get* e *set* para cada variável de instância.

  A maioria destes são em comum com a classe `CommissionEmployee`.


```
1  // Figura 9.6: BaseplusCommissionEmployee.java
2  // A classe BasePlusCommissionEmployee representa um empregado que
3  // recebe um salário-base além da comissão.
4
5  public class BasePlusCommissionEmployee
6  {
7     private String firstName;
8     private String lastName;
9     private String socialSecurityNumber;
10    private double grossSales; // vendas brutas semanais
11    private double commissionRate; // porcentagem da comissão
12    private double baseSalary; // salário-base por semana
13
14    // construtor de seis argumentos
15    public BasePlusCommissionEmployee( String first, String last,
16       String ssn, double sales, double rate, double salary )
17    {
18       // chamada implícita para o construtor Object ocorre aqui
19       firstName = first;
20       lastName = last;
21       socialSecurityNumber = ssn;
22       setGrossSales( sales ); // valida e armazena as vendas brutas
```

As únicas novas informações nos dados da classe `BasePlusCommissionEmployee`

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (Parte 1 de 6.)


```
23          setCommissionRate( rate ); // valida e armazena a taxa de comissão
24          setBaseSalary( salary ); // valida e armazena salário-base
25       } // fim do construtor BasePlusCommissionEmployee de seis argumentos
26
27       // configura o nome
28       public void setFirstName( String first )
29       {
30          firstName = first; // deve validar
31       } // fim do método setFirstName
32
33       // retorna o nome
34       public String getFirstName()
35       {
36          return firstName;
37       } // fim do método getFirstName
38
39       // configura o sobrenome
40       public void setLastName( String last )
41       {
42          lastName = last; // deve validar
43       } // fim do método setLastName
44
```

Inicializa o salário-base

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (Parte 2 de 6.)


```
45       // retorna o sobrenome
46       public String getLastName()
47       {
48          return lastName;
49       } // fim do método getLastName
50
51       // configura o CIC
52       public void setSocialSecurityNumber( String ssn )
53       {
54          socialSecurityNumber = ssn; // deve validar
55       } // fim do método setSocialSecurityNumber
56
57       // retorna o CIC
58       public String getSocialSecurityNumber()
59       {
60          return socialSecurityNumber;
61       } // fim do método getSocialSecurityNumber
62
63       // configura a quantidade de vendas brutas
64       public void setGrossSales( double sales )
65       {
66          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
67       } // fim do método setGrossSales
```

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (Parte 3 de 6.)


```
68
69       // retorna a quantidade de vendas brutas
70       public double getGrossSales()
71       {
72          return grossSales;
73       } // fim do método getGrossSales
74
75       // configura a taxa de comissão
76       public void setCommissionRate( double rate )
77       {
78          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
79       } // fim do método setCommissionRate
80
81       // retorna a taxa de comissão
82       public double getCommissionRate()
83       {
84          return commissionRate;
85       } // fim do método getCommissionRate
86
```

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (Parte 4 de 6.)

```
87        // configura o salário-base
88        public void setBaseSalary( double salary )
89        {
90           baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
91        } // fim do método setBaseSalary
92
93        // retorna o salário-base
94        public double getBaseSalary()
95        {
96           return baseSalary;
97        } // fim do método getBaseSalary
98
99        // calcula os lucros
100       public double earnings()
101       {
102          return baseSalary + ( commissionRate * grossSales );
103       } // fim do método earnings
104
```

Semelhante ao método `earnings` de `CommissionEmployee`

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (Parte 5 de 6.)


```
105       // retorna a representação de String de BasePlusCommissionEmployee
106       @Override // indica que esse método sobrescreve um método e superclasse
107       public String toString()
108       {
109          return String.format(
110             "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f ",
111             "base-salaried commission employee", firstName, lastName,
112             "social security number", socialSecurityNumber,
113             "gross sales", grossSales, "commission rate", commissionRate,
114             "base salary", baseSalary );
115       } // fim do método toString
116    } // fim da classe BasePlusCommissionEmployee
```

Semelhante ao método `toString` de `CommissionEmployee`

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (Parte 6 de 6.)

- A classe `BasePlusCommissionEmployee` não especifica "`extends Object`".

  Estende implicitamente `Object`.
- O construtor de `BasePlusCommissionEmployee` invoca o construtor de `Object` implicitamente.


```
1  // Figura 9.7: BasePlusCommissionEmployeeTest.java
2  // Programa de teste de BasePlusCommissionEmployee.
3
4  public class BasePlusCommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instancia o objeto BasePlusCommissionEmployee
9        BasePlusCommissionEmployee employee =
10          new BasePlusCommissionEmployee(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // obtém os dados do empregado comissionado com salário-base
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n", "First name is",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n", "Last name is",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
```

Figura 9.7 | Programa de teste de `BasePlusCommissionEmployee`. (Parte 1 de 3.)


```
24          System.out.printf( "%s %.2f\n", "Commission rate is",
25             employee.getCommissionRate() );
26          System.out.printf( "%s %.2f\n", "Base salary is",
27             employee.getBaseSalary() );
28
29          employee.setBaseSalary( 1000 ); // configura o salário-base
30
31          System.out.printf( "\n%s:\n\n%s\n",
32             "Updated employee information obtained by toString",
33              employee.toString() );
34       } // fim de main
35    } // fim da classe BasePlusCommissionEmployeeTest
```

Figura 9.7 | Programa de teste de `BasePlusCommissionEmployee`. (Parte 2 de 3.)


```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Figura 9.7 | Programa de teste de `BasePlusCommissionEmployee`. (Parte 3 de 3.)

- Grande parte do código de `BasePlusCommissionEmployee` é semelhante, ou idêntico, ao de `CommissionEmployee`.
- As variáveis de instância `private` `firstName` e `lastName` e os métodos `setFirstName`, `getFirstName`, `setLastName` e `getLastName` são idênticos.

  Ambas as classes também contêm métodos *get* e *set* correspondentes.
- Os construtores são quase idênticos.

  O construtor de `BasePlusCommissionEmployee` também configura `base-Salary`.
- Os métodos `toString` são aproximadamente idênticos.

  O método `toString` de `BasePlusCommissionEmployee` também envia para a saída a variável de instância `baseSalary`.

- Literalmente *copiamos* o código da classe `CommissionEmployee` e o colamos na classe `BasePlusCommissionEmployee`. Então, modificamos a nova classe para incluir um salário-base e métodos que manipulam o salário-base.

  Essa abordagem "copiar e colar" é frequentemente propensa a erro e demorada.

  Ela espalha cópias do mesmo código por todo o sistema, criando um pesadelo para a manutenção de código.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.3

Com a herança, as variáveis de instância comuns e os métodos de todas as classes na hierarquia são declarados em uma superclasse. Quando são feitas modificações nessas características comuns na superclasse, as subclasses herdam, portanto, as modificações. Sem a herança, as alterações precisariam ser feitas em todos os arquivos de código-fonte que contêm uma cópia do código em questão.

9.4.3 Criando uma hierarquia de herança
CommissionEmployee–BasePlusCommissionEmployee

- A classe `BasePlusCommissionEmployee` estende a classe `CommissionEmployee`.
- Um objeto `BasePlusCommissionEmployee` *é um* `CommissionEmployee`.

  Passos da herança nas capacidades da classe `CommissionEmployee`.
- A classe `BasePlusCommissionEmployee` também tem a variável de instância `baseSalary`.
- A subclasse `BasePlusCommissionEmployee` herda as variáveis de instância e métodos de `CommissionEmployee`.

  Apenas os membros `public` e `protected` da superclasse são diretamente acessíveis na subclasse.


```
1  // Figura 9.8: BaseplusCommissionEmployee.java
2  // membros private da superclasse não podem ser acessados em uma subclasse.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // salário-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       // chamada explícita para o construtor CommissionEmployee da superclasse
13       super( first, last, ssn, sales, rate );
14
15       setBaseSalary( salary ); // valida e armazena salário-base
16    } // fim do construtor BasePlusCommissionEmployee de seis argumentos
17
18    // configura o salário-base
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // fim do método setBaseSalary
```

Nova subclasse de CommissionEmployee

Deve-se chamar o construtor da superclasse primeiro

Figura 9.8 | Os membros `private` da superclasse não podem ser acessados em uma subclasse. (Parte 1 de 5.)

```
23
24      // retorna o salário-base
25      public double getBaseSalary()
26      {
27         return baseSalary;
28      } // fim do método getBaseSalary
29
30      // calcula os lucros
31      @Override // indica que esse método sobrescreve um método de superclasse
32      public double earnings()
33      {
34         // não permitido: commissionRate e grossSales private em superclasse
35         return baseSalary + ( commissionRate * grossSales );
36      } // fim do método earnings
37
```

Variáveis de instância `private` de `CommissionEmployee` não são acessíveis aqui

Figura 9.8 | Os membros `private` da superclasse não podem ser acessados em uma subclasse. (Parte 2 de 5.)


```
38      // retorna a representação de String de BasePlusCommissionEmployee
39      @Override // indica que esse método sobrescreve um método de superclasse
40      public String toString()
41      {
42         // não permitido: tenta acessar membros private da superclasse
43         return String.format(
44            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
45            "base-salaried commission employee", firstName, lastName,
46            "social security number", socialSecurityNumber,
47            "gross saless", grossSales, "commission rate", commissionRate,
48            "base salary", baseSalary );
49      } // fim do método toString
50   } // fim da classe BasePlusCommissionEmployee
```

Variáveis de instância `private` de `CommissionEmployee` não são acessíveis aqui

Figura 9.8 | Os membros `private` da superclasse não podem ser acessados em uma subclasse. (Parte 3 de 5.)


```
BasePlusCommissionEmployee.java:35: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee.java:35: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee.java:45: firstName has private access in CommissionEmployee
        "base-salaried commission employee", firstName, lastName,
                                             ^
BasePlusCommissionEmployee.java:45: lastName has private access in CommissionEmployee
        "base-salaried commission employee", firstName, lastName,
                                                        ^
```

Figura 9.8 | Os membros `private` da superclasse não podem ser acessados em uma subclasse. (Parte 4 de 5.)

```
BasePlusCommissionEmployee.java:46: socialSecurityNumber has private access in
CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:47: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee.java:47: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Figura 9.8 | Os membros `private` da superclasse não podem ser acessados em uma subclasse. (Parte 5 de 5.)

- Cada construtor de subclasse deve chamar implícita ou explicitamente seu construtor de superclasse para inicializar as variáveis de instância herdadas da superclasse.

  Sintaxe de chamada de construtor de superclasse — palavra-chave `super`, seguida pelo conjunto de parênteses contendo os argumentos do construtor da superclasse.

  Deve ser a primeira instrução no corpo do construtor da subclasse.
- Se o construtor da subclasse não invocasse o construtor da superclasse explicitamente, o Java tentaria invocar o construtor sem argumentos ou construtor padrão da superclasse.

  A classe `CommissionEmployee` não tem esse construtor, portanto o compilador emitiria um erro.
- Você pode usar explicitamente `super()` para chamar construtor sem argumento ou padrão da superclasse, mas raramente se faz isso.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 9.3

Um erro de compilação ocorre se um construtor de subclasse chamar um construtor de superclasse com argumentos que não coincidem com o número e os tipos de parâmetro em um dos construtores da superclasse.

- Os erros de compilação ocorrem quando a subclasse tenta acessar as variáveis de instância `private` da superclasse.
- Essas linhas também poderiam ter utilizado os métodos *get* adequados para recuperar os valores das variáveis de instância da superclasse.

9.4.4 Hierarquia de herança CommissionEmployee–BasePlusCommissionEmployee utilizando variáveis de instância protected

- Para permitir que uma subclasse acesse diretamente as variáveis de instância da superclasse, podemos declarar esses membros como `protected` na superclasse.
- A nova classe `CommissionEmployee` teve apenas as linhas 6–10 modificadas:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- Com as variáveis de instância `protected`, a subclasse obtém acesso às variáveis de instância, mas as classes que não são subclasses e as classes que não estão no mesmo pacote não podem acessar essas variáveis diretamente.

- A classe `BasePlusCommissionEmployee` (Fig. 9.9) estende a nova versão da classe `CommissionEmployee` com variáveis de instância `protected`.

  Estas variáveis são agora membros `protected` de `BasePlusCommissionEmployee`.
- Se outra classe estender essa versão da classe `BasePlusCommissionEmployee`, a nova subclasse também poderá acessar os membros `protected`.
- O código-fonte na Figura 9.9 (47 linhas) é consideravelmente mais curto do que aquele na Figura 9.6 (116 linhas)

  A maioria das funcionalidades agora são herdadas de `CommissionEmployee`.

  Há agora só uma cópia das funcionalidades.

  Facilita a manutenção, modificação e depuração do código — o código relacionado a um empregado comissionado existe apenas na classe `CommissionEmployee`.


```
1  // Figura 9.9: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee herda a instância protected
3  // variáveis de CommissionEmployee.
4
5  public class BasePlusCommissionEmployee extends CommissionEmployee
6  {
7     private double baseSalary; // salário-base por semana
8
9     // construtor de seis argumentos
10    public BasePlusCommissionEmployee( String first, String last,
11       String ssn, double sales, double rate, double salary )
12    {
13       super( first, last, ssn, sales, rate );
14       setBaseSalary( salary ); // valida e armazena salário-base
15    } // fim do construtor BasePlusCommissionEmployee de seis argumentos
16
17    // configura o salário-base
18    public void setBaseSalary( double salary )
19    {
20       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21    } // fim do método setBaseSalary
22
```

Figura 9.9 | `BasePlusCommissionEmployee` herda as variáveis de instância `protected` de `CommissionEmployee`. (Parte 1 de 3.)

```
23      // retorna o salário-base
24      public double getBaseSalary()
25      {
26         return baseSalary;
27      } // fim do método getBaseSalary
28
29      // calcula os lucros
30      @Override // indica que esse método sobrescreve um método de superclasse
31      public double earnings()
32      {
33         return baseSalary + ( commissionRate * grossSales );
34      } // fim do método earnings
35
36      // retorna a representação de String de BasePlusCommissionEmployee
```

Variáveis de instância `protected` de `CommissionEmployee` *são* acessíveis aqui

Figura 9.9 | `BasePlusCommissionEmployee` herda as variáveis de instância `protected` de `CommissionEmployee`. (Parte 2 de 3.)


```
37      @Override // indica que esse método sobrescreve um método de superclasse
38      public String toString()
39      {
40         return String.format(
41            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
42            "base-salaried commission employee", firstName, lastName,
43            "social security number", socialSecurityNumber,
44            "gross sales", grossSales, "commission rate", commissionRate,
45            "base salary", baseSalary );
46      } // fim do método toString
47   } // fim da classe BasePlusCommissionEmployee
```

Variáveis de instância `private` de `CommissionEmployee` *são* acessíveis aqui

Figura 9.9 | `BasePlusCommissionEmployee` herda as variáveis de instância `protected` de `CommissionEmployee`. (Parte 3 de 3.)

- Herdar as variáveis de instância `protected` aumenta ligeiramente o desempenho, porque podemos acessar diretamente as variáveis na subclasse sem estar sujeitos ao overhead de uma chamada de método *set* ou *get*.
- Na maioria dos casos, é melhor utilizar as variáveis de instância `private` para incentivar a engenharia de software adequada e deixar as questões de otimização de código para o compilador.

  Facilita a manutenção, modificação e depuração do código.

- Utilizar variáveis de instância `protected` cria vários problemas potenciais.
- O objeto de subclasse pode configurar o valor de uma variável herdada diretamente sem utilizar um método *set*.

  Um objeto de subclasse pode atribuir um valor inválido à variável, possivelmente deixando o objeto em um estado inconsistente.
- Métodos de subclasse tendem a ser escritos de modo a depender da implementação de dados da superclasse.

  Subclasses devem depender somente dos serviços da superclasse e não da implementação dos dados da superclasse.

- Com as variáveis de instância `protected` na superclasse, podemos precisar modificar todas as subclasses da superclasse se a implementação de superclasse mudar.

  Nesse caso, diz-se que o software é **frágil** ou **quebradiço**, porque uma pequena alteração na superclasse pode "quebrar" a implementação da subclasse.

  Você deve ser capaz de alterar a implementação de superclasse ao mesmo tempo em que ainda fornece os mesmos serviços às subclasses.

  Se os serviços de superclasse mudam, devemos reimplementar nossas subclasses.
- Os membros `protected` de uma classe são visíveis a todas as classes no mesmo pacote que a classe que contêm os membros `protected` — isso nem sempre é desejável.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.4

Utilize o modificador de acesso `protected` *quando uma superclasse precisar fornecer um método somente para suas subclasses e outras classes no mesmo pacote, mas não para outros clientes.*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.5

Declarar as variáveis de instância da superclasse `private` *(em oposição a* `protected`*) permite a implementação de superclasse dessas variáveis de instância para alterar sem afetar as implementações de subclasse.*

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 9.2

Quando possível, não inclua variáveis de instância `protected` *em uma superclasse. Em vez disso, inclua métodos não* `private` *que acessam as variáveis de instância* `private`*. Isso ajudará a assegurar que os objetos da classe mantenham estados consistentes.*

- Reengenharia da hierarquia usando boas práticas de engenharia de software.
- A classe `CommissionEmployee` declara as variáveis de instância `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` como `private` e fornece métodos `public` para manipular esses valores.

- Os métodos `earnings` e `toString` de `CommissionEmployee` utilizam os métodos *get* da classe da classe para obter os valores de suas variáveis de instância.

  Se você decidir alterar a representação interna dos dados, somente o corpo dos métodos *get* e *set* que manipulam diretamente as variáveis de instância precisarão mudar.

  Essas modificações só ocorrem dentro da superclasse — nenhuma modificação para a subclasse é necessária.

  Localizar os efeitos de alterações como esta é uma boa prática de engenharia de software.
- A subclasse `BasePlusCommissionEmployee` herda métodos não `private` de `Commission-Employee` e pode acessar os membros `private` da superclasse via esses métodos.


```
1  // Figura 9.10: CommissionEmployee.java
2  // A classe CommissionEmployee utiliza métodos para manipular suas
3  // variáveis de instância private.
4  public class CommissionEmployee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // vendas brutas semanais
10    private double commissionRate; // porcentagem da comissão
11
12    // construtor de cinco argumentos
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // chamada implícita para o construtor Object ocorre aqui
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // valida e armazena as vendas brutas
21       setCommissionRate( rate ); // valida e armazena a taxa de comissão
22    } // fim do construtor CommissionEmployee de cinco argumentos
23
```

Os dados são `private` para melhor encapusalamento; torna o código mais fácil de manter/depurar

Figura 9.10 | A classe `CommissionEmployee` utiliza métodos para manipular suas variáveis de instância `private`. (Parte 1 de 5.)


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

Figura 9.10 | A classe `CommissionEmployee` utiliza métodos para manipular suas variáveis de instância `private`. (Parte 2 de 5.)

```
47
48       // configura o CIC
49       public void setSocialSecurityNumber( String ssn )
50       {
51          socialSecurityNumber = ssn; // deve validar
52       } // fim do método setSocialSecurityNumber
53
54       // retorna o CIC
55       public String getSocialSecurityNumber()
56       {
57          return socialSecurityNumber;
58       } // fim do método getSocialSecurityNumber
59
60       // configura a quantidade de vendas brutas
61       public void setGrossSales( double sales )
62       {
63          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64       } // fim do método setGrossSales
65
66       // retorna a quantidade de vendas brutas
67       public double getGrossSales()
68       {
```

Figura 9.10 | A classe `CommissionEmployee` utiliza métodos para manipular suas variáveis de instância `private`. (Parte 3 de 5.)

```
69          return grossSales;
70       } // fim do método getGrossSales
71
72       // configura a taxa de comissão
73       public void setCommissionRate( double rate )
74       {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76       } // fim do método setCommissionRate
77
78       // retorna a taxa de comissão
79       public double getCommissionRate()
80       {
81          return commissionRate;
82       } // fim do método getCommissionRate
83
84       // calcula os lucros
85       public double earnings()
86       {
87          return getCommissionRate() * getGrossSales();
88       } // fim do método earnings
89
```

Não mais acessando diretamente variáveis de instância aqui

Figura 9.10 | A classe `CommissionEmployee` utiliza métodos para manipular suas variáveis de instância `private`. (Parte 4 de 5.)


```
90      // retorna a representação String do objeto CommissionEmployee
91      @Override // indica que esse método sobrescreve um método de superclasse
92      public String toString()
93      {
94         return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
95            "commission employee", getFirstName(),getLastName(),
96            "social security number", getSocialSecurityNumber(),
97            "gross sales", getGrossSales(),
98            "commission rate", getCommissionRate() );
99      } // fim do método toString
100  } // fim da classe CommissionEmployee
```

Não mais acessando diretamente variáveis de instância aqui

Figura 9.10 | A classe `CommissionEmployee` utiliza métodos para manipular suas variáveis de instância `private`. (Parte 5 de 5.)

- A classe `BasePlusCommissionEmployee` (Figura 9.11) tem várias modificações que a distinguem da Figura 9.9.
- Os métodos `earnings` e `toString` invocam suas versões de superclasse e não acessam variáveis de instância diretamente.


```
1  // Figura 9.11: BaseplusCommissionEmployee.java
2  // A classe BasePlusCommissionEmployee herda de CommissionEmployee
3  // e acessa os dados private da superclasse via
4  // métodos public herdados.
5
6  public class BasePlusCommissionEmployee extends CommissionEmployee
7  {
8     private double baseSalary; // salário-base por semana
9
10    // construtor de seis argumentos
11    public BasePlusCommissionEmployee( String first, String last,
12       String ssn, double sales, double rate, double salary )
13    {
14       super( first, last, ssn, sales, rate );
15       setBaseSalary( salary ); // valida e armazena salário-base
16    } // fim do construtor BasePlusCommissionEmployee de seis argumentos
17
18    // configura o salário-base
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // fim do método setBaseSalary
```

Figura 9.11 | A classe `BasePlusCommissionEmployee` herda de `CommissionEmployee` e acessa os dados `private` da superclasse via métodos `public` herdados. (Parte 1 de 2.)


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

Figura 9.11 | A classe `BasePlusCommissionEmployee` herda de `CommissionEmployee` e acessa os dados `private` da superclasse via métodos `public` herdados. (Parte 2 de 2.)

- O método `earnings` sobrescreve o método `earnings`.
- A nova versão chama o método `earnings` de `CommissionEmployee` com `super.earnings()`.

  Obtém o lucro com base na comissão apenas.
- colocar a palavra-chave `super` e um ponto (`.`) separador antes do nome de método de superclasse invoca a versão de superclasse de um método sobrescrito.
- Boa prática de engenharia de software.

  Se um método realizar todas ou algumas ações necessárias por outro método, chame esse método em vez de duplicar seu código.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 9.4

Quando um método de superclasse é sobrescrito em uma subclasse, a versão de subclasse frequentemente chama a versão de superclasse para fazer uma parte do trabalho. A falha em prefixar o nome do método da superclasse com a palavra-chave `super` *e um ponto separador* (`.`) *ao chamar o método da superclasse faz com que o método de subclasse chame a si próprio, criando potencialmente um erro chamado recursão infinita. A recursão, utilizada corretamente, é uma capacidade poderosa discutida no Capítulo 18, "Recursão".*

- O método `toString` de `BasePlusCommissionEmployee` sobrescreve o método `toString` da classe `CommissionEmployee.`
- A nova versão cria parte da representação `String` chamando o método `toString` de `CommissionEmployee` com a expressão `super.toString()`.

9.5 Construtores em subclasses

- Instanciar um objeto de subclasse inicia uma cadeia de chamadas de construtor.
  O construtor de subclasse, antes de realizar suas próprias tarefas, invoca o construtor da sua superclasse direta.
- Se a superclasse é derivada de outra classe, o construtor de superclasse invoca o construtor da próxima classe acima na hierarquia, e assim por diante.
- O último construtor chamado na cadeia é sempre o construtor da classe `Object`.
- O corpo do construtor de subclasse original termina a execução por último.
- O construtor de cada superclasse manipula as variáveis de instância de superclasse que o objeto de subclasse herda.

Observação de engenharia de software 9.6

Quando um programa cria um objeto de subclasse, o construtor de subclasse imediatamente chama o construtor de superclasse (explicitamente, via `super`*, ou implicitamente). O corpo do construtor de superclasse executa para inicializar as variáveis de instância da superclasse que fazem parte do objeto de subclasse, então o corpo do construtor de subclasse executa para inicializar variáveis de instância somente de subclasse. O Java assegura que mesmo se um construtor não atribuir um valor a uma variável de instância, a variável ainda é inicializada como seu valor padrão (por exemplo,* `0` *para tipos numéricos primitivos,* `false` *para* `boolean`*s,* `null` *para referências).*

9.6 Engenharia de software com herança

- Ao estender uma classe, a nova classe herda os membros da superclasse — embora os membros `private` da superclasse permaneçam ocultos na nova classe.
- Você pode personalizar a nova classe para atender a suas necessidades incluindo membros adicionais e sobrescrevendo membros de superclasse.

  Fazer isso não requer que o programador da subclasse altere o (ou mesmo tenha acesso ao) código-fonte da superclasse.

  O Java simplesmente requer acesso ao arquivo `.class` da superclasse.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.7

Embora herdar de uma classe não requeira acesso ao código-fonte da classe, os desenvolvedores frequentemente insistem em examinar o código-fonte para entender como a classe é implementada. Os desenvolvedores na indústria querem assegurar que eles estejam estendendo uma classe sólida — por exemplo, uma classe que executa bem e é implementada de maneira firme e segura.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.8

Na etapa do design de um sistema orientado a objetos, você frequentemente descobrirá que certas classes estão intimamente relacionadas. Você deve "fatorar" as variáveis de instância e métodos comuns e colocá-los em uma superclasse. Depois, utilize a herança para desenvolver subclasses, especializando-as com capacidades além das herdadas da superclasse.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.9

Declarar uma subclasse não afeta o código-fonte da sua superclasse. A herança preserva a integridade da superclasse.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 9.10

Assim como os projetistas de sistemas não orientados a objetos devem evitar a proliferação de métodos, os projetistas de sistemas orientados a objetos devem evitar a proliferação de classes. Essa proliferação cria problemas de gerenciamento e pode prejudicar a capacidade de reutilização de software, porque em uma enorme biblioteca de classes torna-se difícil localizar as classes mais apropriadas. A alternativa é criar menos classes que fornecem funcionalidades mais substanciais, mas isso pode se tornar complicado.

Java™ COMO PROGRAMAR 8ª edição

Dica de desempenho 9.1

Se as subclasses são maiores do que precisam ser (isto é, contêm funcionalidades demais), recursos de memória e de processamento podem ser desperdiçados. Estenda a superclasse que contém a funcionalidade mais próxima daquilo de que você precisa.

9.7 Classe `Object`

- Todas as classes do Java herdam direta ou indiretamente da classe `Object`; portanto, seus 11 métodos são herdados por todas as outras classes.
- A Figura 9.12 resume os métodos de `Object`.
- Você pode aprender mais sobre métodos `Object` na documentação on-line da API e em *The Java Tutorial at :*

  `java.sun.com/javase-/6/docs/api/java/lang/Object.html`

  ou

  `java.sun.com/docs/books/tutorial/java/IandI/objectclass.html`
- Todo array tem um método `clone` sobrescrito que copia o array.

  Se o array armazenar referências a objetos, os objetos não serão copiados — uma cópia superficial é realizada.
- Para mais informações sobre o relacionamento entre arrays e a classe `Object`, consulte *Java Language Specification, Capítulo 10, em*

  `java.sun.com/docs/books/jls/third_edition/html/arrays.html`

Método	Descrição
`clone`	Esse método `protected`, que não aceita nenhum argumento e retorna uma referência `Object`, faz uma cópia do objeto em que é chamado. A implementação padrão realiza a chamada **cópia superficial** — os valores da variável de instância em um objeto são copiados em outro objeto do mesmo tipo. Para tipos por referência, apenas as referências são copiadas. Uma típica implementação do método `clone` sobrescrito realizaria uma **cópia em profundidade** que cria um novo objeto para cada variável de instância de tipo por referência. É difícil implementar `clone` corretamente. Por isso, seu uso não é recomendável. Muitos especialistas do setor sugerem utilizar a serialização de objetos em seu lugar. Discutimos a serialização de objetos no Capítulo 17, "Arquivos, fluxos e serialização de objetos".
`equals`	Esse método compara dois objetos quanto à igualdade e retorna `true` se eles forem iguais ou, caso contrário, `false`. O método aceita qualquer `Object` como um argumento. Quando os objetos de uma classe particular precisam ser comparados quanto à igualdade, a classe deve sobrescrever o método `equals` para comparar o *conteúdo* dos dois objetos.

Figura 9.12 | Métodos de `Object`. (Parte 1 de 3.)

Método	Descrição
`equals` (*cont.*)	Para os requisitos da implementação desse método, consulte a documentação do método em `java.sun.com/javase/6/docs/api/java/lang/Object.html# equals(java.lang.Object)`. A implementação padrão de `equals` usa o operador `==` para determinar se duas referências *referenciam o mesmo objeto* na memória. A Seção 16.3.3 demonstra o método `equals` da classe `String` e diferencia entre comparar objetos `String` com `==` e com `equals`.
`finalize`	Esse método `protected` (introduzido na Seção 8.10) é chamado pelo coletor de lixo para realizar a limpeza de término em um objeto antes de o coletor de lixo reivindicar a memória do objeto. Lembre-se de que não é claro se, ou quando, o método `finalize` será chamado. Por essa razão, a maioria dos programadores deve evitar o método `finalize`.
`getClass`	Todo objeto no Java conhece seu próprio tipo em tempo de execução. O método `getClass` (utilizado nas seções 10.5, 14.5 e 24.3) retorna um objeto de classe `Class` (pacote `java.lang`) que contém as informações sobre o tipo de objeto, como seu nome de classe (retornado pelo método `Class` `getName`). Para obter informações adicionais sobre a classe `Class`, visite `java.sun.com/javase/6/docs/api/java/lang/Class.html`.

Figura 9.12 | Métodos de `Object`. (Parte 2 de 3.)

Método	Descrição
`hashCode`	Códigos de hash são valores `int` que são úteis para armazenamento e recuperação de alta velocidade das informações armazenadas em uma estrutura de dados que é conhecida como uma tabela de hash (discutida na Seção 20.11). Esse método também é chamado como parte da implementação padrão do método `toString` da classe `Object`.
`wait`, `notify`, `notifyAll`	Os métodos `notify`, `notifyAll` e três versões sobrecarregadas de `wait` estão relacionados ao multithreading, discutido no Capítulo 26.
`toString`	Esse método (introduzido na Seção 9.4.1) retorna uma representação `String` de um objeto. A implementação padrão desse método retorna o nome de pacote e o nome de classe da classe do objeto seguido por uma representação hexadecimal do valor retornado pelo método `hashCode` do objeto.

Figura 9.12 | Métodos de `Object`. (Parte 3 de 3.)

9.8 (Opcional) Estudo de caso de GUI e imagens gráficas: exibindo texto e imagens com rótulos

- **Rótulos** são um modo conveniente de identificar componentes GUI na tela e manter o usuário informado sobre o estado atual do programa.
- Um **JLabel** (do pacote `javax.swing`) pode exibir texto, uma imagem ou ambos.
- O exemplo na Figura 9.13 demonstra vários recursos `JLabel`, incluindo um rótulo de texto sem formatação, um rótulo de imagem e um rótulo tanto com texto como com uma imagem.


```
1  // Figura 9.13: LabelDemo.java
2  // Demonstra o uso de rótulos.
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
5  import javax.swing.JLabel;
6  import javax.swing.JFrame;
7
8  public class LabelDemo
9  {
10    public static void main( String[] args )
11    {
12       // Cria um rótulo com texto simples
13       JLabel northLabel = new JLabel( "North" );
14
15       // cria um ícone de uma imagem para podermos colocar em um JLabel
16       ImageIcon labelIcon = new ImageIcon( "GUItip.gif" );
17
18       // cria um rótulo com um Icon em vez de texto
19       JLabel centerLabel = new JLabel( labelIcon );
20
21       // cria outro rótulo com um Icon
22       JLabel southLabel = new JLabel( labelIcon );
23
```

Figura 9.13 | `JLabel` com texto e imagens. (Parte 1 de 3.)


```
24          // configura o rótulo para exibir texto (bem como um ícone)
25          southLabel.setText( "South" );
26
27          // cria um frame para armazenar os rótulos
28          JFrame application = new JFrame();
29
30          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
31
32          // adiciona os rótulos ao frame; o segundo argumento especifica
33          // onde adicionar o rótulo no frame
34          application.add( northLabel, BorderLayout.NORTH );
35          application.add( centerLabel, BorderLayout.CENTER );
36          application.add( southLabel, BorderLayout.SOUTH );
37
38          application.setSize( 300, 300 ); // configura o tamanho do frame
39          application.setVisible( true ); // mostra o frame
40       } // fim de main
41    } // fim da classe LabelDemo
```

Figura 9.13 | `JLabel` com texto e imagens. (Parte 2 de 3.)

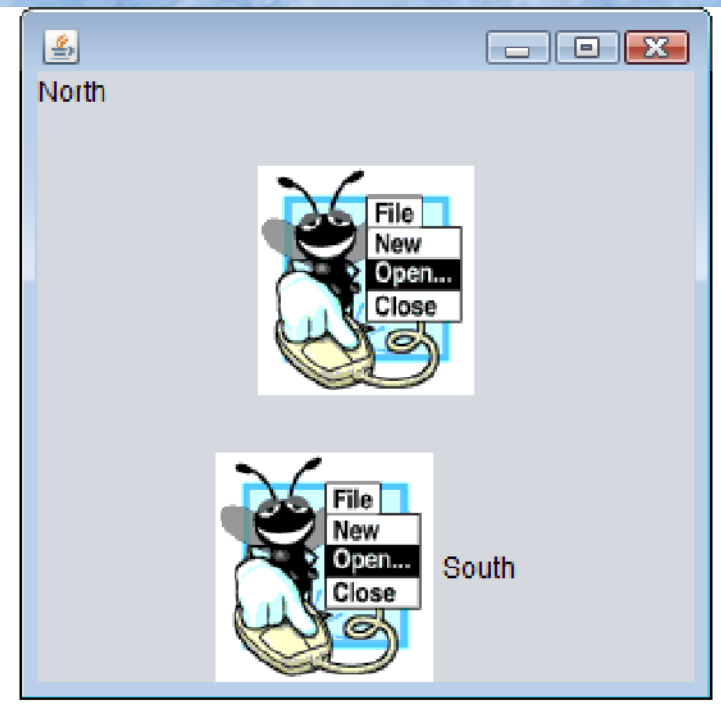


Figura 9.13 | `JLabel` com texto e imagens. (Parte 3 de 3.)

- Um **ImageIcon** representa uma imagem que pode ser exibida em um `JLabel`.
- O construtor para `ImageIcon` recebe uma `String` que especifica o caminho para a imagem.
- `ImageIcon` possa carregar imagens nos formatos de imagem GIF, JPEG e PNG.
- O método **setText** de `JLabel` muda o texto que o rótulo exibe na tela.

- Uma versão sobrecarregada do método `add` que recebe dois parâmetros permite especificar o componente GUI para adicionar a um `JFrame` e a localização em que adicioná-lo.
  - O primeiro parâmetro é o componente a anexar.
  - O segundo é a região em que ele deve ser colocado.
- Cada `JFrame` tem um **layout** para posicionar componentes GUI.
  - O layout default de um `JFrame` é **BorderLayout.**
  - Cinco regiões — `NORTH` (no alto), `SOUTH` (embaixo), `EAST` (à direita), `WEST` (à esquerda) e `CENTER` (constantes na classe `BorderLayout`).
  - Cada uma dessas regiões é declarada como uma constante na classe `BorderLayout`.
- Ao chamar o método `add` com um argumento, o `JFrame` coloca o componente no `CENTER` do `BorderLayout` automaticamente.