10

Programação orientada a objetos: Polimorfismo

OBJETIVOS

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

Sumário

Sumário

10.1 Introdução

- **Polimorfismo:**
  - **Permite ‘programação no geral’.**
  - **A mesma invocação pode produzir ‘muitas formas’ de resultados.**
- **Interfaces:**
  - **Implementadas pelas classes a fim de atribuir funcionalidades comuns a classes possivelmente não-relacionadas.**

10.2 Exemplos de polimorfismo

- **Polimorfismo:**
  - **Quando um programa invoca um método por meio de uma variável de superclasse, a versão correta de subclasse do método é chamada com base no tipo da referência armazenada na variável da superclasse.**
  - **Com o polimorfismo, o mesmo nome e assinatura de método podem ser utilizados para fazer com que diferentes ações ocorram, dependendo do tipo de objeto em que o método é invocado.**
  - **Facilita a adição de novas classes a um sistema com o mínimo de modificações no código do sistema.**

Observação de engenharia de software 10.1

O polimorfismo permite que programadores tratem de generalidades e deixem que o ambiente de tempo de execução trate as especificidades. Os programadores podem instruir objetos a se comportarem de maneiras apropriadas para esses objetos, sem nem mesmo conhecer os tipos dos objetos (contanto que os objetos pertençam à mesma hierarquia de herança).

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 exigir modificações no sistema básico. Somente o código de cliente que instancia os novos objetos deve ser modificado para, assim, acomodar os novos tipos.

10.3 Demonstrando um comportamento polimórfico

- **Uma referência de superclasse pode ter por alvo um objeto de subclasse:**
  - **Isso é possível porque um objeto de subclasse também *é um* objeto de superclasse.**
  - **Ao invocar um método a partir dessa referência, o tipo do *objeto referenciado real*, não o tipo da *referência*, determina qual método é chamado.**
- **Uma referência de subclasse pode ter por alvo um objeto de superclasse somente se o objeto sofrer *downcasting*.**

Resumo

PolymorphismTest

.java

```
1  // Fig. 10.1: PolymorphismTest.java
2  // Atribuindo referências de superclasse e subclasse a variáveis de superclasse e
3  // 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       CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // atribui uma referência de subclasse a variável de subclasse
14       BasePlusCommissionEmployee4 basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee4(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoca toString na superclasse object usando a variável de superclasse
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Call CommissionEmployee3's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
23       // invoca toString no objeto de subclasse usando a variável de subclasse
24       System.out.printf( "%s %s:\n\n%s\n\n",
25          "Call BasePlusCommissionEmployee4's toString with subclass",
26          "reference to subclass object",
27          basePlusCommissionEmployee.toString() );
28
```

Atribuições de referência típicas

(1 de 2)

```
29       // invoca toString no objeto de subclasse usando a
30       CommissionEmployee3 commissionEmployee2 =
31          basePlusCommissionEmployee;
32       System.out.printf( "%s %s:\n\n%s\n",
33          "Call BasePlusCommissionEmployee4's toString with superclass",
34          "reference to subclass object", commissionEmployee2.toString() );
35    } // fim de main
36 } // fim da classe PolymorphismTest
```

Atribui uma referência a um objeto **basePlusCommissionEmployee** a uma variável **CommissionEmployee3**

PolymorphismTest

Chama polimorficamente o método **toString** de **basePlusCommissionEmployee**

```
Call CommissionEmployee3's toString with superclas
object:

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

Call BasePlusCommissionEmployee4'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 BasePlusCommissionEmployee4'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
```


10.4 Classes e métodos abstratos

- **Classes abstratas:**
  - **Classes que são demasiadamente gerais para criar objetos reais.**
  - **Utilizadas somente como superclasses abstratas para subclasses concretas e para declarar variáveis de referência.**
  - **Muitas hierarquias de herança têm superclasses abstratas que ocupam os poucos níveis superiores.**
  - **Palavra-chave `abstract`:**
    - **Utilize para declarar uma classe `abstract`.**
    - **Também utilize para declarar um método `abstract`:**
      - **As classes abstratas normalmente contêm um ou mais métodos abstratos.**
      - **Todas as subclasses concretas devem sobrescrever todos os métodos abstratos herdados.**

10.4 Classes e métodos abstratos (*Continuação*)

- **Classe `Iteradora`:**
  - **Pode percorrer todos os objetos em uma coleção, como um array ou um `ArrayList`.**
  - **Os iteradores são freqüentemente utilizados na programação polimórfica para percorrer uma coleção que contém referências a objetos provenientes de vários níveis de uma hierarquia.**

Observação de engenharia de software 10.3

Uma classe abstrata declara atributos e comportamentos comuns 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 as subclasses precisarem ser concretas. Variáveis de instância e métodos concretos de uma classe abstrata estão sujeitos às regras normais da herança.

Erro comum de programação 10.1

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

Erro de programação comum 10.2

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

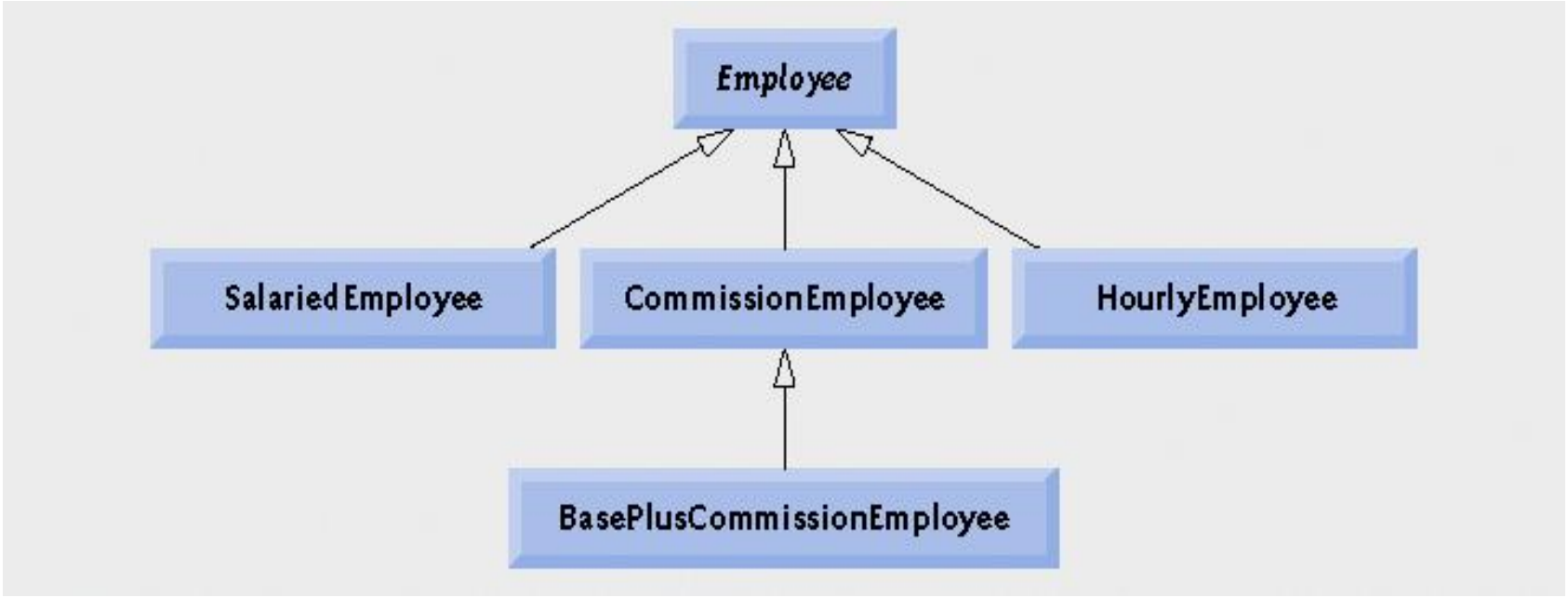

Figura 10.2 | Diagrama de classes UML da hierarquia `Employee`.

Observação de engenharia de software 10.4

Uma subclasse pode herdar a ‘interface’ ou ‘implementação’ de uma superclasse. Hierarquias projetadas para a `herança de implementação` tendem a ter suas funcionalidades na parte superior da hierarquia — cada nova subclasse herda um ou mais métodos que foram implementados em uma superclasse e a subclasse utiliza essas implementações de superclasse. (*Continua*…)

Observação de engenharia de software 10.4 (*Continuação*)

As hierarquias projetadas para a herança de `interface` tendem a ter suas funcionalidades na parte inferior da hierarquia — uma superclasse especifica um ou mais métodos abstratos que devem ser declarados para cada classe concreta na hierarquia; e as subclasses individuais sobrescrevem esses métodos para fornecer implementações específicas de subclasses.

10.5.1 Criando a superclasse abstrata Employee

- Superclasse `abstract` `Employee`:
  - `earnings` é declarado `abstract`.
    - Nenhuma implementação pode ser dada a `earnings` na classe `abstract` `Employee`.
  - Um array de variáveis `Employee` armazenará as referências a objetos de subclasse.
    - Chamadas ao método `earnings` a partir dessas variáveis chamarão a versão apropriada do método `earnings`.

	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 *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`.

Resumo

```
1  // Fig. 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
```

Declara a classe **Employee** como **abstract**

Atributos comuns a todos os empregados

oyee.java

(1 de 3)

```
18    // configura o nome
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // fim do método setFirstName
23
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;
34    } // fim do método setLastName
35
36    // retorna o sobrenome
37    public String getLastName()
38    {
39       return lastName;
40    } // fim do método getLastName
41
```

Resumo

Employee.java

(2 de 3)

Resumo

Employee.java

(3 de 3)

```
42    // configura CIC
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // fim do método setSocialSecurityNumber
47
48    // retorna 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    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // fim do método toString
60
61    // método abstrato sobrescrito pelas subclasses
62    public abstract double earnings(); // nenhuma implementação aqui
63 } // fim da classe Employee abstrata
```

Método **`abstract earnings`** não tem nenhuma implementação

Resumo

SalariedEmployee.java

(1 de 2)

```
1  // Fig. 10.5: SalariedEmployee.java
2  // Classe SalariedEmployee estende Employee.
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 salário
14    } // fim do construtor SalariedEmployee constructor com quatro argumentos
15
16    // configura salário
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // fim do método setWeeklySalary
21
```

Classe **SalariedEmployee** estende a classe **Employee**

Chama construtor de superclasse

Chama o método **setWeeklySalary**

Valida e configura o valor do salário semanal

Resumo

SalariedEmployee.java (2 de 2)

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

Sobrescreve o método **earnings** para que **SalariedEmployee** possa ser concreta

Sobrescreve o método **toString**

Chama a versão do **toString** da superclasse

Resumo

HourlyEmployee.java

(1 de 2)

```
1  // Fig. 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 por 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
24    // retorna a remuneração
25    public double getWage()
26    {
27       return wage;
28    } // fim do método getWage
29
```

Classe **HourlyEmployee** estende a classe **Employee**

Chama construtor de superclasse

Valida e configura o valor do salário por hora

Resumo

HourlyEmployee .java

(2 de 2)

```
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 horas trabalhadas
38    public double getHours()
39    {
40       return hours;
41    } // fim do método getHours
42
43    // calcula lucros; sobrescreve o método earnings em Employee
44    public double earnings()
45    {
46       if ( getHours() <= 40 ) // nenhuma hora extra
47          return getWage() * getHours();
48       else
49          return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50    } // fim do método earnings
51
52    // retorna a representação de String do objeto HourlyEmployee
53    public String toString()
54    {
55       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
56          super.toString(), "hourly wage", getWage(),
57          "hours worked", getHours() );
58    } // fim do método toString
59 } // fim da classe HourlyEmployee
```

Valida e configura o valor das horas trabalhadas

Sobrescreve o método **earnings** para que **HourlyEmployee** possa ser concreta

Sobrescreve o método **toString**

Chama o método **toString** da superclasse

Resumo

CommissionEmployee.java

(1 de 3)

```
1  // Fig. 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
```

Classe **CommissionEmployee** estende a classe **Employee**

Chama construtor de superclasse

Valida e configura o valor da taxa de comissão

Resumo

CommissionEmployee.java

(2 de 3)

```
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 brut
37    public double getGrossSales()
38    {
39       return grossSales;
40    } // fim do método getGrossSales
41
```

Valida e configura o valor das vendas brutas

Resumo

```
42    // calcula os rendimentos; sobrescreve o método earnings em Employee
43    public double earnings()
44    {
45       return getCommissionRate() * getGrossSales();
46    } // fim do método earnings
47
48    // retorna a representação String do objeto CommissionEmployee
49    public String toString()
50    {
51       return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
52          "commission employee", super.toString(),
53          "gross sales", getGrossSales(),
54          "commission rate", getCommissionRate() );
55    } // fim do método toString
56 } // fim da classe CommissionEmployee
```

Sobrescreve o método **earnings** para que **CommissionEmployee** possa ser concreta

CommissionEmployee.java

Sobrescreve o método **toString**

(3 de 3)

Chama o método **toString** da superclasse

Resumo

BasePlusCommission Employee.java

(1 de 2)

```
1  // Fig. 10.8: BasePlusCommissionEmp
2  // Classe BasePlusCommissionEmploye
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       super( first, last, ssn, sales, rate );
13       setBaseSalary( salary ); // valida e armazena s
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
```

A classe **BasePlusCommissionEmployee** estende a classe CommissionEmployee

Chama construtor de superclasse

Valida e configura o valor do salário-base

Resumo

```
22    // retorna 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    public double earnings()
30    {
31       return getBaseSalary() + super.earnings();
32    } // fim do método earnings
33
34    // retorna representação de String do objeto BasePlusCommissionEmployee
35    public String toString()
36    {
37       return String.format( "%s %s; %s: $%,.2f",
38          "base-salaried", super.toString(),
39          "base salary", getBaseSalary() );
40    } // fim do método toString
41 } // fim da classe BasePlusCommissionEmployee
```

BasePlusCommissionEmployee.java

Sobrescreve o método **earnings**

Chama o método **earnings** da superclasse

e 2)

Sobrescreve o método **toString**

Chama o método **toString** da superclasse

Resumo

PayrollSystemTest.java

(1 de 5)

```
1  // Fig. 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 da 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
```

Resumo

PayrollSystemTest.java

(2 de 5)

```
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23          salariedEmployee, "earned", salariedEmployee.earnings() );
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
```

Atribuindo objetos de subclasse a variáveis de superclasse

Chama implícita e polimorficamente `toString`

Resumo

```
48          // determina se elemento é um BasePlusCommissionEmployee
49          if ( currentEmployee instanceof BasePlusCommissionEmployee )
50          {
51             // downcast da referência de Employee par
52             // a referência BasePlusCommissionEmploye
53             BasePlusCommissionEmployee employee =
54                ( BasePlusCommissionEmployee ) currentEmployee;
55
56             double oldBaseSalary = employee.getBaseSalary();
57             employee.setBaseSalary( 1.10 * oldBaseSalary );
58             System.out.printf(
59                "new base salary with 10%% increase is: $%,.2f\n",
60                employee.getBaseSalary() );
61          } // fim de if
62
63          System.out.printf(
64             "earned $%,.2f\n\n", currentEmployee.earnings() );
65       } // fim de for
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
```

Se a variável `currentEmployee` apontar para um objeto `BasePlusCommissionEmployee`

`PayrollSystemTest`

Downcast de **`CurrentEmployee`** em uma referência a `BasePlusCommissionEmployee`

(3 de 5)

Dá a `BasePlusCommissionEmployees` um bônus de 10% em relação ao salário-base

Chama polimorficamente o método `earnings`

Chama os métodos **`getClass`** e `getName` a fim de exibir o nome de classe do objeto de cada subclasse **`Employee`**

```
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
```

Resumo

PayrollSystemTest

.java

(4 de 5)

Resumo

```
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

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

Mesmos resultados que ao processar os empregados individualmente

O salário-base é aumentado em 10%

Cada tipo de empregado é exibido

`PayrollSystemTest.java`

(5 de 5)

10.5.6 Demonstrando o processamento polimórfico, o operador `instanceof` e o downcasting

- **Vinculação dinâmica:**
  - **Também conhecida como vinculação tardia.**
  - **Chamadas aos métodos sobrescritos, elas são resolvidas em tempo de execução com base no tipo de objeto referenciado.**
- **Operador `instanceof`:**
  - **Determina se um objeto é uma instância de certo tipo.**

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.

Observação de engenharia de software 10.5

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 coerção 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.

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 um objeto no seu próprio tipo ou no tipo de uma das suas superclasses.

10.5.6 Demonstrando o processamento polimórfico, o operador `instanceof` e o downcasting (*Continuação*)

- **Downcasting:**
  - **Converte uma referência a uma superclasse em uma referência a uma subclasse.**
  - **Permitido somente se o objeto tiver um relacionamento *é um* com a subclasse.**
- **Método `getClass`:**
  - **Herdado de `Object`.**
  - **Retorna um objeto do tipo `Class`.**
- **Método `getName` da classe `Class`:**
  - **Retorna o nome da classe.**

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

- **Regras de atribuição 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 por causa do relacionamento *é um*.**
    - **Referenciar membros exclusivos de subclasses por meio de variáveis de superclasse é um erro de compilação.**
  - **Atribuir uma referência de superclasse a uma variável de subclasse é um erro de compilação.**
    - **O downcasting pode evitar esse erro.**

10.6 Métodos e classes `final`

- **Métodos `final`:**
  - **Não podem ser sobrescritos em uma subclasse.**
  - **Métodos `private` e `static` são implicitamente `final`.**
  - **Métodos `final` são resolvidos em tempo de compilação, isso é conhecido como vinculação estática.**
    - **Os compiladores podem otimizar colocando o código em linha.**
- **Classes `final`:**
  - **Não podem ser estendidas por uma subclasse.**
  - **Todos os métodos em uma classe `final` são implicitamente `final`.**

Dica de desempenho 10.1

O compilador pode decidir fazer uma inclusão inline de uma chamada de método `final` e fará isso para métodos `final` pequenos e simples. A inclusão inline não viola o encapsulamento nem o ocultamento de informações, mas aprimora o desempenho porque elimina o overhead de fazer uma chamada de método.

Erro comum de programação 10.5

Tentar declarar uma subclasse de uma classe `final` é um erro de compilação.

Observação de engenharia de software 10.6

Na API do Java, a ampla maioria das classes não é declarada `final`. Isso permite a herança e o polimorfismo — as capacidades fundamentais da programação orientada a objetos. Entretanto, em alguns casos, é importante declarar classes `final` em geral por questões de segurança.

10.7 Estudo de caso: Criando e utilizando interfaces

- **Interfaces:**
  - **Palavra-chave `interface`.**
  - **Contém somente constantes e métodos `abstract`:**
    - **Todos os campos são implicitamente `public`, `static` e `final`.**
    - **Todos os métodos são métodos `abstract` implicitamente `public`.**
  - **Classes podem `implement`ar interfaces:**
    - **A classe deve declarar cada método na interface utilizando a mesma assinatura ou a classe deve ser declarada `abstract`.**
  - **Em geral, utilizada quando diferentes classes precisam compartilhar métodos e constantes comuns.**
  - **Normalmente, declaradas nos seus próprios arquivos com os mesmos nomes das interfaces e com a extensão de nome de arquivo `.java`.**

Boa prática de programação 10.1

De acordo com o Capítulo 9, *Especificação de linguagem Java*, 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 maneira semelhante, as constantes devem ser declaradas sem as palavras-chave `public`, `static` e `final` porque elas também são redundantes.

Erro comum de programação 10.6

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

10.7.1 Desenvolvendo uma hierarquia Payable

- **Interface `Payable`:**
  - **Contém o método `getPaymentAmount`.**
  - **É implementada pelas classes `Invoice` e `Employee`.**
- **Representação UML das interfaces:**
  - **Distinguimos as interfaces das classes colocando a palavra 'interface' entre aspas francesas (« e ») acima do nome da interface.**
  - **O relacionamento entre uma classe e uma interface é conhecido como *realização***
    - **Uma classe 'realiza', ou implementa, o método de uma interface.**

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 um amplo intervalo de classes não-relacionadas.

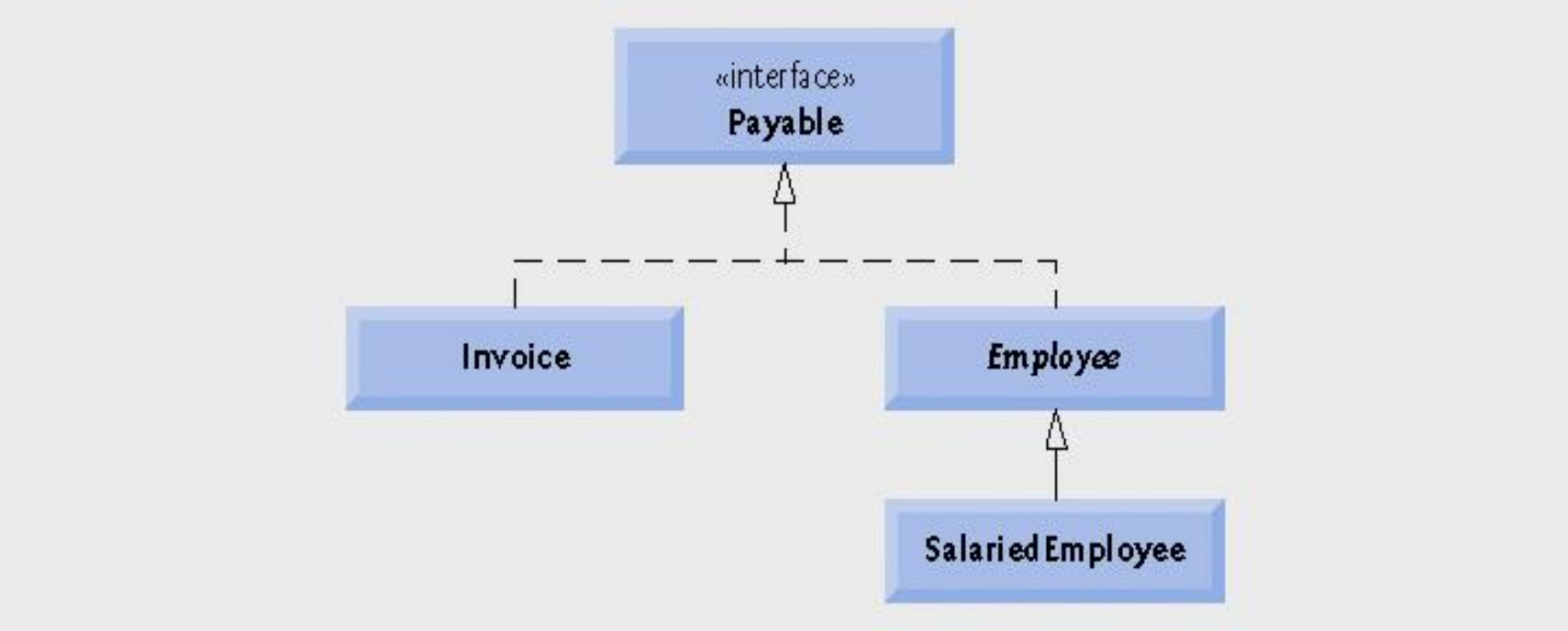

Figura 10.10 | Diagrama de classe UML da hierarquia `Payable`.

Resumo

Payable.java

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

Declara a interface **Payable**

Declara o método **getPaymentAmount** que é implicitamente **public** e **abstract**

Resumo

Invoice.java

(1 de 3)

```
1  // Fig. 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 de 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 ); // validate and store price per item
19    } // fim do construtor Invoice de quatro argumentos
20
21    // configura número de peças
22    public void setPartNumber( String part )
23    {
24       partNumber = part;
25    } // fim do método setPartNumber
26
```

A classe **Invoice** implementa a interface **Payable**

Resumo

Invoice.java

(2 de 3)

```
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;
37    } // fim do método setPartDescription
38
39    // obtém a descrição
40    public String getPartDescription()
41    {
42       return partDescription;
43    } // fim do método getPartDescription
44
45    // configura quantidade
46    public void setQuantity( int count )
47    {
48       quantity = ( count < 0 ) ? 0 : count; // 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
```

Resumo

Invoice.java

(3 de 3)

```
57    // configura preço por item
58    public void setPricePerItem( double price )
59    {
60       pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
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
69    // retorna a representação de String do objeto Invoice
70    public String toString()
71    {
72       return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73          "invoice", "part number", getPartNumber(), getPartDescription(),
74          "quantity", getQuantity(), "price per item", getPricePerItem() );
75    } // fim do método toString
76
77    // método requerido para executar o contrato com a interface Payable
78    public double getPaymentAmount()
79    {
80       return getQuantity() * getPricePerItem(); // calcula o custo total
81    } // fim do método getPaymentAmount
82 } // fim da classe Invoice
```

Declara **getPaymentAmount** para cumprir o contrato com a interface **Payable**

10.7.3 Criando a classe `Invoice`

- **Uma classe pode implementar quantas interfaces precisar:**
  - **Utilize uma lista separada por vírgulas dos nomes de interfaces depois da palavra-chave `implements`.**
    - **Exemplo: `public class` *NomeDaClasse* `extends` *NomeDaSuperclasse* `implements` *PrimeiraInterface*, *SegundaInterface*, …**

Resumo

Employee.java

(1 de 3)

```
1  // Fig. 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 de 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 de três argumentos
17
```

A classe **Employee** implementa a interface **Payable**

Resumo

Employee.java

(2 de 3)

```
18    // configura o nome
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // fim do método setFirstName
23
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;
34    } // fim do método setLastName
35
36    // retorna o sobrenome
37    public String getLastName()
38    {
39       return lastName;
40    } // fim do método getLastName
41
```

Resumo

Employee.java

(3 de 3)

```
42    // configura o CIC
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // deve validar
46    } // fim do método setSocialSecurityNumber
47
48    // retorna 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    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // fim do método toString
60
61    // Nota: Não implementamos o método getPaymentAmount de Payable aqui, assim
62    // esta classe deve ser declarada abstrata para evitar um erro de compilação.
63 } // fim da classe Employee abstrata
```

O método **getPaymentAmount** não é implementado aqui

10.7.5 Modificando a classe `SalariedEmployee` para uso na hierarquia `Payable`

- **Os objetos de quaisquer subclasses da classe que implementa a interface também podem ser pensados como objetos da interface.**
  - **Uma referência a um objeto de subclasse pode ser atribuída a uma variável de interface se a superclasse implementar essa interface.**

Observação de engenharia de software 10.7

Herança e interfaces são semelhantes quanto à sua implementação do relacionamento ‘é um’. Um objeto de uma classe que implementa uma interface pode ser pensado como um objeto desse tipo de interface. Um objeto de quaisquer subclasses de uma classe que implementa uma interface também pode ser pensado como um objeto do tipo de interface.

```java
// Fig. 10.14: SalariedEmployee.java
// Classe SalariedEmployee estende Employee, que implementa Payable.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // construtor de quatro argumentos
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // passa para o construtor Employee
      setWeeklySalary( salary ); // valida e armazena salário
   } // fim do construtor SalariedEmployee de quatro argumentos

   // configura salário
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // fim do método setWeeklySalary

```

A classe **SalariedEmployee** estende a classe **Employee** (que implementa a interface **Payable**)

Resumo

SalariedEmployee.java

(1 de 2)

Resumo

SalariedEmployee.java

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

Declara o método **getPaymentAmount** em vez do método **earnings**

(2 de 2)

Observação de engenharia de software 10.8

O relacionamento ‘é um’ que ocorre entre superclasses e subclasses, e entre interfaces e as classes que as implementam, é mantido ao passar um objeto para um método. Quando um parâmetro de método recebe uma variável de uma superclasse ou tipo de interface, o método processa o objeto recebido polimorficamente como um argumento.

Observação de engenharia de software 10.9

Utilizando uma referência de superclasse, podemos invocar polimorficamente qualquer método especificado na declaração de superclasse (e na classe `Object`). Utilizando uma referência de interface, podemos invocar polimorficamente qualquer método especificado na declaração de interface (e na classe `Object`).

Resumo

PayableInterfaceTest.java

```
1  // Fig. 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
```

Declara um array de variáveis **Payable**

Atribuindo referências a objetos **Invoice** para variáveis **Payable**

Atribuindo referências a objetos **SalariedEmployee** para variáveis **Payable**

Resumo

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

```
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
```

Chama os métodos **`toString`** e **`getPaymentAmount`** polimorficamente

`PayableInterfaceTest.java`

(2 de 2)

Observação de engenharia de software 10.10

Todos os métodos da classe `Object` podem ser chamados utilizando uma referência de um tipo de interface. Uma referência referencia um objeto e todos os objetos herdam os métodos da classe `Object`.

10.7.7 Declarando constantes com interfaces

- **Interfaces podem ser utilizadas para declarar constantes utilizadas em muitas declarações de classes:**
  - **Essas constantes são implicitamente `public`, `static` e `final`.**
  - **Utilizar uma declaração `static import` permite que os clientes utilizem essas constantes apenas com seus nomes.**

Observação de engenharia de software 10.11

A partir do J2SE 5.0, uma melhor prática de programação é criar conjuntos de constantes como enumerações com a palavra-chave `enum`. Consulte a Seção 6.10 para uma introdução a `enum` e a Seção 8.9 para detalhes adicionais sobre `enum`.

Interface	Descrição
`Comparable`	Como aprendeu no Capítulo 2, o Java contém vários operadores de comparação (por exemplo, <, <=, >, >=, ==, !=) que permitem comparar valores primitivos. Entretanto, esses operadores não podem ser utilizados para comparar o conteúdo dos objetos. A interface `Comparable` é utilizada para permitir que objetos de uma classe, que implementam a interface, sejam comparados entre si. A interface contém um método, `compareTo`, que compara o objeto que chama o método com o objeto passado como um argumento para o método. As classes devem implementar o `compareTo` para que ele retorne uma quantia indicando se o objeto em que é invocado é menor que (valor de retorno inteiro negativo), igual a (valor de retorno `0`) ou maior que (valor de retorno inteiro positivo) o objeto passado como um argumento, utilizando quaisquer critérios especificados pelo programador. Por exemplo, se classe `Employee` implementar `Comparable`, seu método `compareTo` poderia comparar objetos `Employee` de acordo com suas quantias de vencimentos. A interface `Comparable` é comumente utilizada para ordenar objetos em uma coleção, como um array. Utilizamos `Comparable` no Capítulo 18,Genéricos, e no Capítulo 19, Coleções.
`Serializable`	Uma interface de tags utilizada somente para identificar classes cujos objetos podem ser gravados (isto é, serializados) ou lidos de (isto é, desserializados) algum tipo de armazenamento (por exemplo, arquivo em disco, campo de banco de dados) ou transmitidos por uma rede. Utilizamos `Serializable` no Capítulo 14, Arquivos e fluxos.

Figura 10.16 | Interfaces comuns da API do Java. (Parte 1 de 2.)

Interface	Descrição
`Runnable`	Implementas por qualquer classe por meio das quais objetos dessa classe devem ser capazes de executar em paralelo utilizando uma técnica chamada multithreading (discutida no Capítulo 23, Multithreading). A interface contém um método, `run`, que descreve o comportamento de um objeto quando executado.
Interfaces ouvintes de eventos GUI	Você utiliza interfaces gráficas com o usuário (GUIs) todos os dias. Por exemplo, no seu navegador da Web, você digitaria em um campo de texto o endereço de um site da Web que quer visitar ou clicaria em um botão para retornar ao site anterior que visitou. Ao digitar o endereço de um site da Web ou clicar em um botão no navegador da Web, o navegador deve responder à sua interação e realizar 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 11, Componentes GUI: Parte 1 e o Capítulo 22, Componentes GUI: Parte 2, você aprenderá a criar GUIs em Java e como construir handlers de eventos para responder a interações de usuário. Os handlers de eventos são declarados em classes que implementam uma interface ouvinte de evento 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 11 e 22.

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

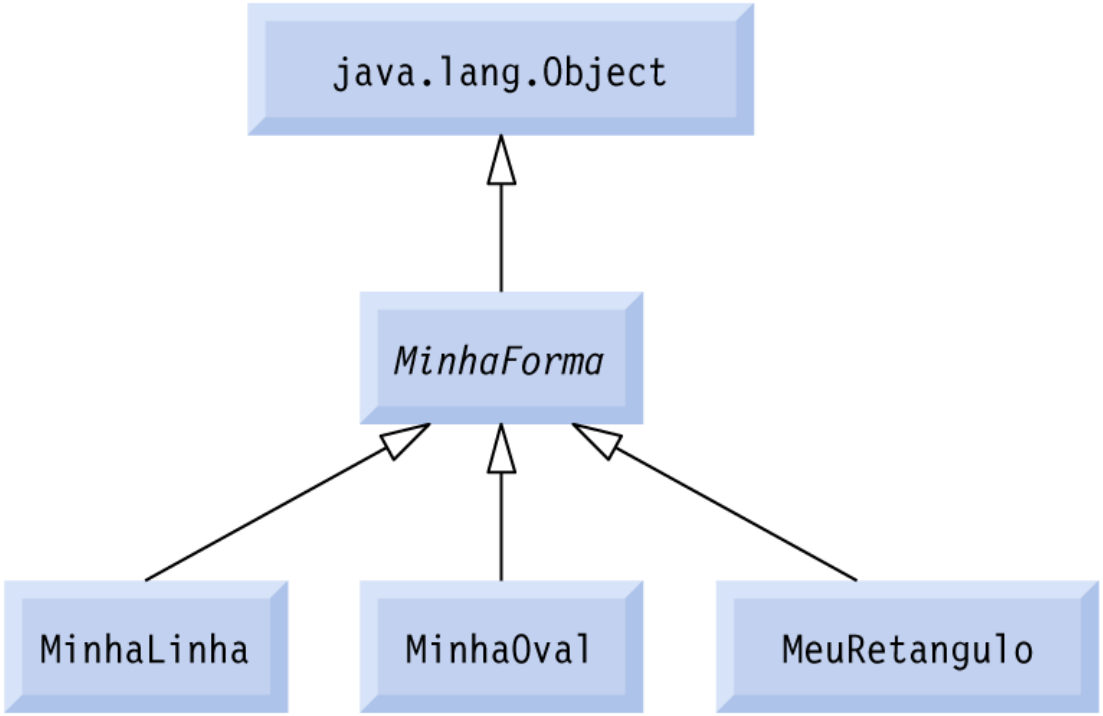

Figura 10.17 | Hierarquia de `MinhaForma`.

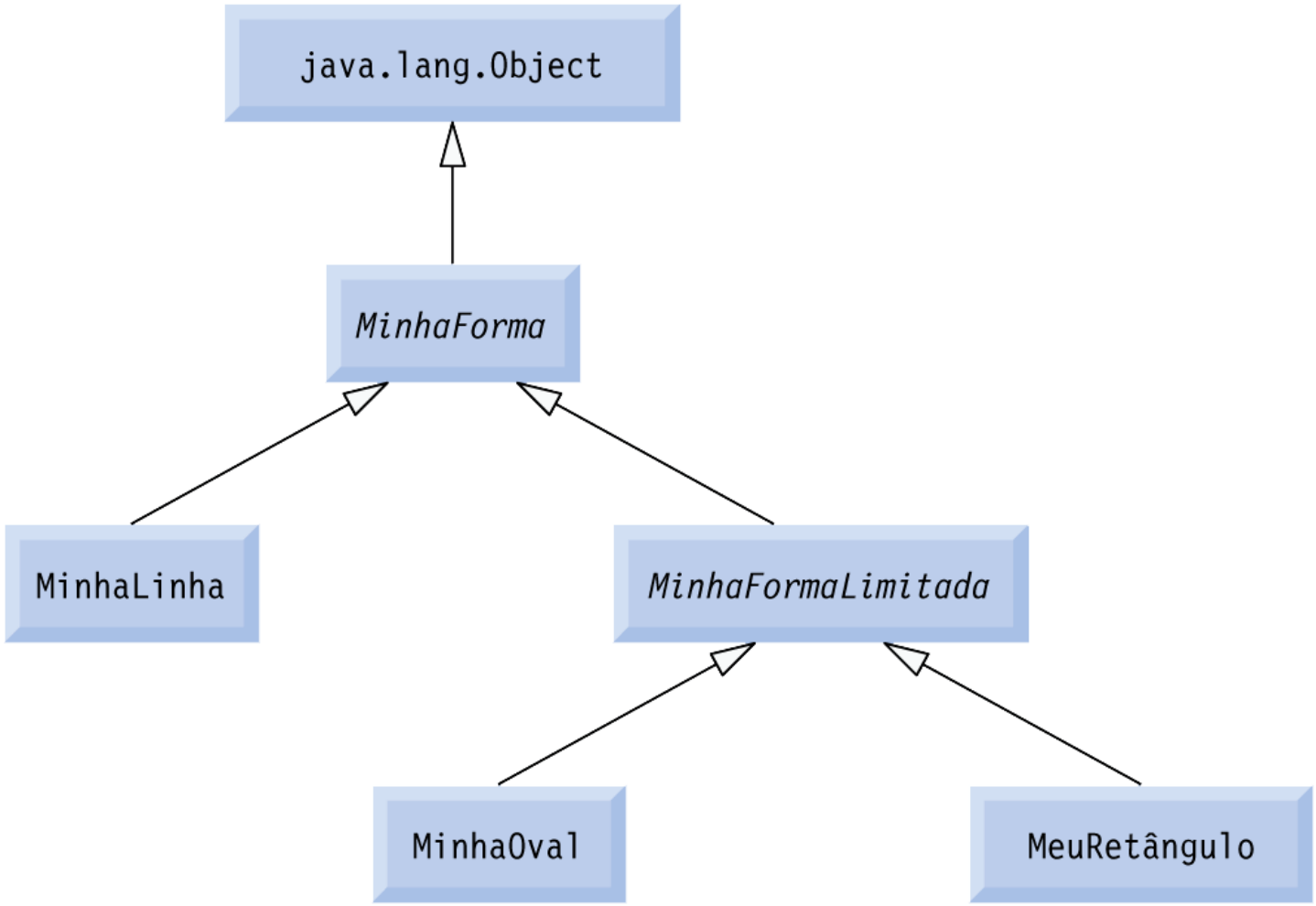

Figura 10.18 | Hierarquia de `MinhaForma` com `MinhaFormaLimitada`.

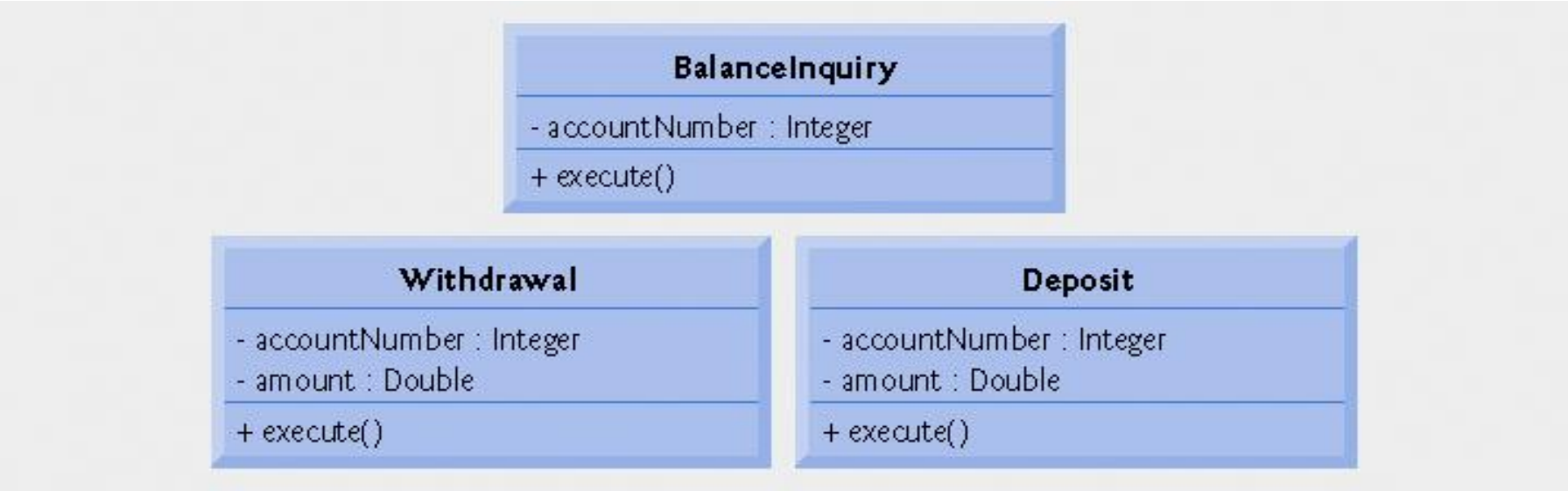

Figura 10.19 | Atributos e operações das classes `BalanceInquiry`, `Withdrawal` e `Deposit`.

10.9 (Opcional) Estudo de caso de engenharia de software: Incorporando herança ao sistema ATM

- **Modelo UML para herança:**
  - **Relacionamento de generalização.**
    - **A superclasse é uma generalização das subclasses.**
    - **As subclasses são especializações da superclasse.**
- **Superclasse `Transaction`:**
  - **Contém os métodos e campos que `BalanceInquiry`, `Withdrawal` e `Deposit` têm em comum.**
    - **Método `execute`.**
    - **Campo `accountNumber`.**

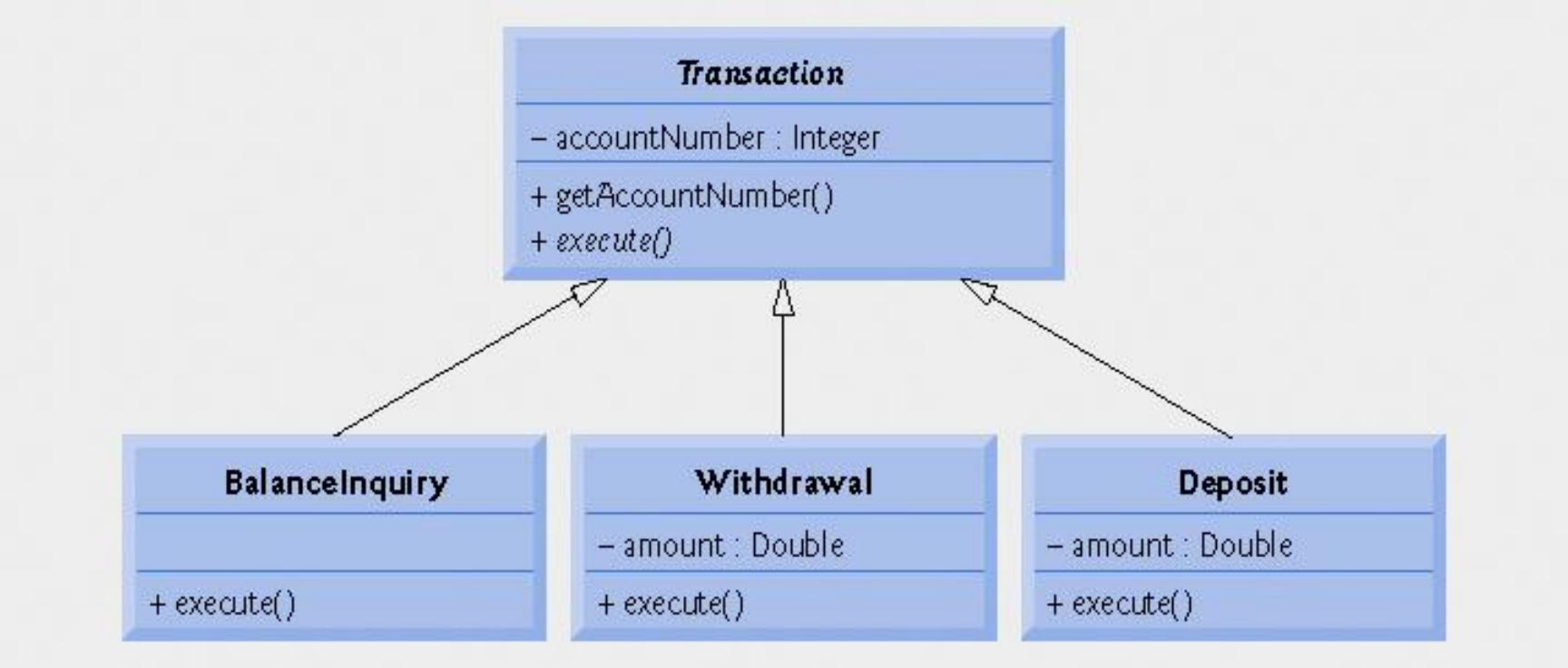

Figura 10. 20 | Diagrama de classes que modela a generalização da superclasse `Transaction` e das subclasses `BalanceInquiry`, `Withdrawal` e `Deposit`. Observe que os nomes das classes abstratas (por exemplo, `Transaction`) e os nomes dos métodos (por exemplo, `execute` na classe `Transaction`) aparecem em itálico.

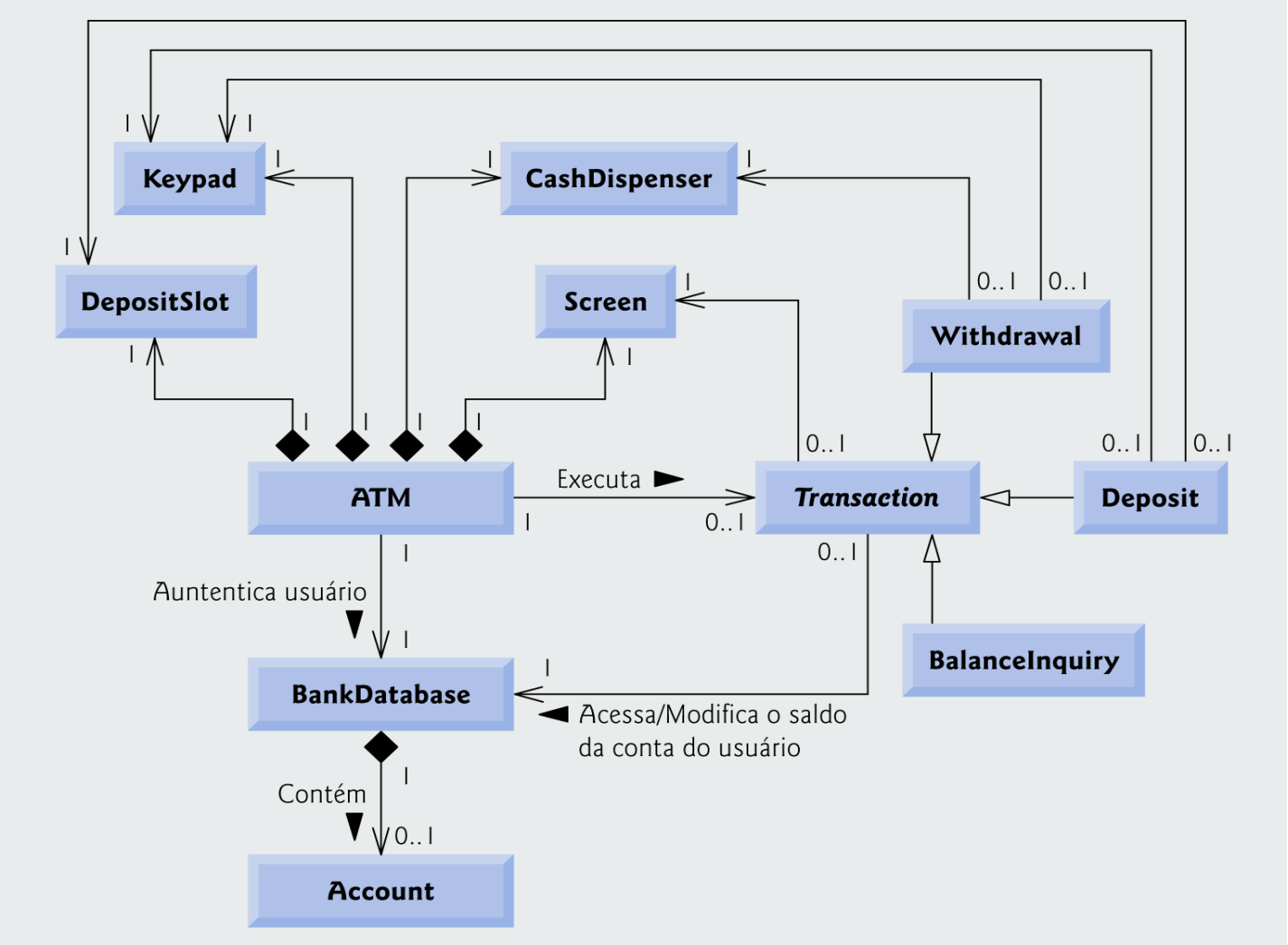

Figura 10.21 | Diagrama de classes do sistema ATM (incorporando herança). Observe que os nomes das classes abstratas (por exemplo, Transaction) aparecem em itálico .

Observação de engenharia de software 10.12

Um diagrama de classes completo mostra todas as associações entre as classes e todos os atributos e operações para cada classe. Se o número de atributos de classe, métodos e associações for muito alto (como nas figuras 10.21 e 10.22), uma boa prática que promove a legibilidade é dividir essas informações entre dois diagramas de classes — um focalizando as associações e, o outro, os atributos e métodos.

10.9 (Opcional) Estudo de caso de engenharia de software: Incorporando herança ao sistema ATM (*Continuação*)

- **Incorporando herança ao projeto do sistema ATM:**
  - **Se a classe A for uma generalização da classe B, então a classe B estenderá a classe A.**
  - **Se classe A for uma classe abstrata e a classe B for uma subclasse da classe A, a classe B deve então implementar os métodos abstratos da classe A se a classe B tiver de ser uma classe concreta.**

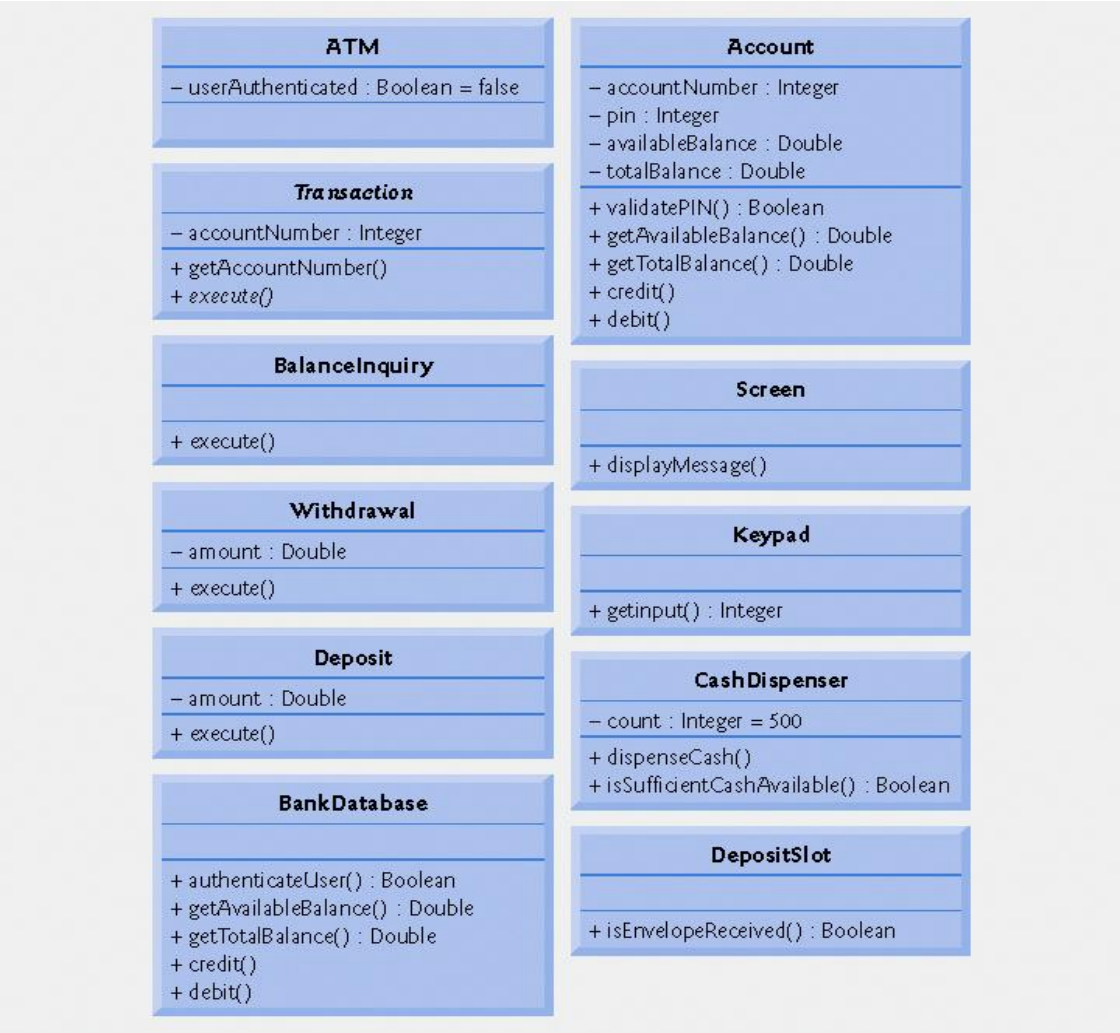

Figura 10.22 | Diagrama de classes com atributos e operações (incorporando herança). Observe que os nomes das classes abstratas (por exemplo, `Transaction`) e o nome do método (por exemplo, `execute` na classe `Transaction`) aparecem em itálico.

Resumo

```
1 // Classe Withdrawal representa uma transação de saque no ATM
2 public class Withdrawal extends Transaction
3 {
4 } // fim da classe Withdrawal
```

A subclasse **`Withdrawal`** estende a superclasse **`Transaction`**

`Withdrawal.java`

Resumo

```
1  // Withdrawal.java
2  // Gerada com os diagramas de classes nas Fig. 10.21 e Fig. 10.22
3  public class Withdrawal extends Transaction
4  {
5     // atributos
6     private double amount; // quantia a retirar
7     private Keypad keypad; // referência ao teclado numérico
8     private CashDispenser cashDispenser; // referência ao dispensador de cédulas
9
10    // construtor sem argumentos
11    public Withdrawal()
12    {
13    } // fim do construtor Withdrawal sem argumentos
14
15    // método sobrescrevendo execute
16    public void execute()
17    {
18    } // fim do método execute
19 } // fim da classe Withdrawal
```

A subclasse **`Withdrawal`** estende a superclasse **`Transaction`**

`thdrawal.jav`
`a`

Observação de engenharia de software 10.13

Várias ferramentas de modelagem da UML convertem projetos baseados em UML em código Java e podem acelerar o processo de implementação consideravelmente.

Para informações adicionais sobre essas ferramentas, consulte Recursos na Internet e na Web listados no final da Seção 2.9.

Resumo

```
1  // Classe abstrata Transaction representa uma transação do ATM
2  public abstract class Transaction
3  {
4     // atributos
5     private int accountNumber; // indi
6     private Screen screen; // Tela do ATM
7     private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
8
9     // construtor sem argumentos invocado pelas subclasses utilizando super()
10    public Transaction()
11    {
12    } // fim do construtor Transaction sem argumentos
13
14    // retorna o número de conta
15    public int getAccountNumber()
16    {
17    } // fim do método getAccountNumber
18
```

Declara a superclasse **abstract** `Transaction`

`Transaction.java`

(1 de 2)

Resumo

Transaction.java

(2 de 2)

```
19    // retorna a referência à tela
20    public Screen getScreen()
21    {
22    } // fim do método getScreen
23
24    // retorna a referência ao banco de dados da instituição financeira
25    public BankDatabase getBankDatabase()
26    {
27    } // fim do método getBankDatabase
28
29    // método abstrato sobrescrito pela subclasses
30    public abstract void execute();
31 } // fim da classe Transaction
```

Declara o método **abstract** **execute**