Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, of methods that a program can invoke on all `Employee` objects. We use the term ' face" here in a general sense to refer to the various ways programs can communicat objects of any `Employee` subclass. Be careful not to confuse the general notion of an ' face" to something with the formal notion of a Java interface, the subject of Sectior Each employee, regardless of the way his or her earnings are calculated, has a first n last name and a social security number, so `private` instance variables `firstName`, `Name` and `socialSecurityNumber` appear in abstract superclass `Employee`.

Software Engineering Observation 10.4

A subclass can inherit "interface" or "implementation" from a superclass. Hierarchies a for implementation inheritance tend to have their functionality high in the hierarchy new subclass inherits one or more methods that were implemented in a superclass, a subclass uses the superclass implementations. Hierarchies designed for interface inheritan to have their functionality lower in the hierarchy—a superclass specifies one or more a methods that must be declared for each concrete class in the hierarchy, and the ina subclasses override these methods to provide subclass-specific implementations.

The following sections implement the `Employee` class hierarchy. The first fou tions each implement one of the concrete classes. The last section implements a te gram that builds objects of all these classes and processes those objects polymorphi

10.5.1 Creating Abstract Superclass `Employee`

Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition *get* and *set* methods that manipulate `Employee`'s instance variables. An `earnings` m certainly applies generically to all employees. But each earnings calculation depends employee's class. So we declare `earnings` as `abstract` in superclass `Employee` becaus fault implementation does not make sense for that method—there is not enough inf tion to determine what amount `earnings` should return. Each subclass overrides ear with an appropriate implementation. To calculate an employee's earnings, the progr signs a reference to the employee's object to a superclass `Employee` variable, then in the `earnings` method on that variable. We maintain an array of `Employee` variable of which holds a reference to an `Employee` object (of course, there cannot be `Employ` jects because `Employee` is an abstract class—because of inheritance, however, all obj all subclasses of `Employee` may nevertheless be thought of as `Employee` objects). Th gram iterates through the array and calls method `earnings` for each `Employee` objec processes these method calls polymorphically. Including `earnings` as an abstract m in `Employee` forces every direct subclass of `Employee` to override `earnings` in order come a concrete class. This enables the designer of the class hierarchy to demand tha subclass provide an appropriate pay calculation.

Method `toString` in class `Employee` returns a `String` containing the first nam name and social security number of the employee. As we will see, each subcl `Employee` overrides method `toString` to create a string representation of an object class that contains the employee's type (e.g., `"salaried employee:"`) followed by th of the employee's information.

	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*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

shows the desired results of each method. [*Note:* We do not list superclass `Employee`'s *get* and *set* methods because they are not overridden in any of the subclasses—each of these methods is inherited and used "as is" by each of the subclasses.]

Let us consider class `Employee`'s declaration (Fig. 10.4). The class includes a constructor that takes the first name, last name and social security number as arguments (lines 11–16); *get* methods that return the first name, last name and social security number (lines 25–28, 37–40 and 49–52, respectively); *set* methods that set the first name, last name and social security number (lines 19–22, 31–34 and 43–46, respectively); method `toString` (lines 55–59), which returns the string representation of `Employee`; and `abstract` method `earnings` (line 62), which will be implemented by subclasses. Note that the `Employee` constructor does not validate the social security number in this example. Normally, such validation should be provided.

Why did we decide to declare `earnings` as an `abstract` method? It simply does not make sense to provide an implementation of this method in class `Employee`. We cannot calculate the earnings for a general `Employee`—we first must know the specific `Employee` type to determine the appropriate earnings calculation. By declaring this method `abstract`, we indicate that each concrete subclass *must* provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

```
1  // Fig. 10.4: Em
2  // Employee abst
3
4  public abstract
5  {
6     private Strin
7     private Strin
8     private Strin
9
10    // three-argu
11    public Employ
12    {
13       firstName =
14       lastName =
15       socialSecu
16    } // end thre
17
18    // set first
19    public void se
20    {
21       firstName =
22    } // end metho
23
24    // return firs
25    public String
26    {
27       return firs
28    } // end metho
29
30    // set last na
31    public void se
32    {
33       lastName =
34    } // end metho
35
36    // return last
37    public String
38    {
39       return last
40    } // end metho
41
42    // set social
43    public void se
44    {
45       socialSecur
46    } // end metho
47
48    // return soci
49    public String
50    {
51       return soci
52    } // end metho
```

Fig. 10.4 | Employee abs

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last;
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
```

Fig. 10.4 | Employee abstract superclass. (Part 1 of 2.)

```
53
54     // return String representation of Employee object
55     public String toString()
56     {
57        return String.format( "%s %s\nsocial security number: %s",
58           getFirstName(), getLastName(), getSocialSecurityNumber() );
59     } // end method toString
60
61     // abstract method overridden by subclasses
62     public abstract double earnings(); // no implementation here
63  } // end abstract class Employee
```

Fig. 10.4 | Employee abstract superclass. (Part 2 of 2.)

10.5.2 Creating Concrete Subclass SalariedEmployee

Class SalariedEmployee (Fig. 10.5) extends class Employee (line 4) and overrides earnings (lines 29–32), which makes SalariedEmployee a concrete class. The class includes a constructor (lines 9–14) that takes a first name, a last name, a social security number and a weekly salary as arguments; a *set* method to assign a new non-negative value to instance variable weeklySalary (lines 17–20); a *get* method to return weeklySalary's value (lines 23–26); a method earnings (lines 29–32) to calculate a SalariedEmployee's earnings; and a method toString (lines 35–39), which returns a String including the employee's type, namely, "salaried employee: " followed by employee-specific information produced by superclass Employee's toString method and SalariedEmployee's getWeeklySalary method. Class SalariedEmployee's constructor passes the first name, last name and social security number to the Employee constructor (line 12) to initialize the

```
1  // Fig. 10.5: SalariedEmployee.java
2  // SalariedEmployee class extends Employee.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // pass to Employee constructor
13       setWeeklySalary( salary ); // validate and store salary
14    } // end four-argument SalariedEmployee constructor
15
16    // set salary
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // end method setWeeklySalary
21
```

Fig. 10.5 | SalariedEmployee class derived from Employee. (Part 1 of 2.)

```
22    // retu
23    public
24    {
25       retu
26    } // en
27
28    // calc
29    public
30    {
31       retu
32    } // en
33
34    // retu
35    public
36    {
37       retu
38          s
39    } // en
40 } // end c
```

Fig. 10.5 | Sala

private instan
abstract metho
the SalariedEm
riedEmployee
of course, we w

Method to
method toStri
ployee would h
ployee's toStr
security numbe
a complete strin
returns "salari
(i.e., first name
class's toString
of a SalariedEm
the class's getWe

10.5.3 Creati

Class HourlyEmp
constructor (line
number, an hou
set methods that
od setWage (lin
31–35) ensures t
HourlyEmployee
wage and hours,
ployee's earning

```
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; override abstract method earnings in Employee
29    public double earnings()
30    {
31       return getWeeklySalary();
32    } // end method earnings
33
34    // return String representation of SalariedEmployee object
35    public String toString()
36    {
37       return String.format( "salaried employee: %s\n%s: $%,.2f",
38          super.toString(), "weekly salary", getWeeklySalary() );
39    } // end method toString
40 } // end class SalariedEmployee
```

Fig. 10.5 | `SalariedEmployee` class derived from `Employee`. (Part 2 of 2.)

`private` instance variables not inherited from the superclass. Method `earnings` overrides abstract method `earnings` in `Employee` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` must be declared `abstract`—otherwise, a compilation error occurs (and, of course, we want `SalariedEmployee` here to be a concrete class).

Method `toString` (lines 35–39) of class `SalariedEmployee` overrides `Employee` method `toString`. If class `SalariedEmployee` did not override `toString`, `SalariedEmployee` would have inherited the `Employee` version of `toString`. In that case, `SalariedEmployee`'s `toString` method would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To produce a complete string representation of a `SalariedEmployee`, the subclass's `toString` method returns `"salaried employee: "` followed by the superclass `Employee`-specific information (i.e., first name, last name and social security number) obtained by invoking the superclass's `toString` (line 38)—this is a nice example of code reuse. The string representation of a `SalariedEmployee` also contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` method.

10.5.3 Creating Concrete Subclass `HourlyEmployee`

Class `HourlyEmployee` (Fig. 10.6) also extends class `Employee` (line 4). The class includes a constructor (lines 10–16) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked. Lines 19–22 and 31–35 declare *set* methods that assign new values to instance variables `wage` and `hours`, respectively. Method `setWage` (lines 19–22) ensures that `wage` is non-negative, and method `setHours` (lines 31–35) ensures that `hours` is between 0 and 168 (the total number of hours in a week). Class `HourlyEmployee` also includes *get* methods (lines 25–28 and 38–41) to return the values of `wage` and `hours`, respectively; a method `earnings` (lines 44–50) to calculate an `HourlyEmployee`'s earnings; and a method `toString` (lines 53–58), which returns the employee's type,

namely, "hourly employee: " and employee-specific information. Note that the HourlyEmployee constructor, like the SalariedEmployee constructor, passes the first name, last name

```
1  // Fig. 10.6: HourlyEmployee.java
2  // HourlyEmployee class extends Employee.
3
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // wage per hour
7     private double hours; // hours worked for week
8
9     // five-argument constructor
10    public HourlyEmployee( String first, String last, String ssn,
11       double hourlyWage, double hoursWorked )
12    {
13       super( first, last, ssn );
14       setWage( hourlyWage ); // validate hourly wage
15       setHours( hoursWorked ); // validate hours worked
16    } // end five-argument HourlyEmployee constructor
17
18    // set wage
19    public void setWage( double hourlyWage )
20    {
21       wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22    } // end method setWage
23
24    // return wage
25    public double getWage()
26    {
27       return wage;
28    } // end method getWage
29
30    // set hours worked
31    public void setHours( double hoursWorked )
32    {
33       hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34          hoursWorked : 0.0;
35    } // end method setHours
36
37    // return hours worked
38    public double getHours()
39    {
40       return hours;
41    } // end method getHours
42
43    // calculate earnings; override abstract method earnings in Employee
44    public double earnings()
45    {
46       if ( getHours() <= 40 ) // no overtime
47          return getWage() * getHours();
48       else
49          return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50    } // end method earnings
```

Fig. 10.6 | HourlyEmployee class derived from Employee. (Part 1 of 2.)

```
51
52       // retu
53       public
54       {
55          retu
56             s
57             "
58       } // en
59    } // end c
```

Fig. 10.6 | Hour

and social securi
private instance
(line 56) to obta
security number)

10.5.4 Creati

Class Commissio
a constructor (lir
a sales amount ar
values to instanc
(lines 25–28 and
ings (lines 43–4

```
1  // Fig. 10.
2  // Commissi
3
4  public clas
5  {
6     private
7     private
8
9     // five-a
10    public Co
11       double
12    {
13       super(
14       setGro
15       setCom
16    } // end
17
18    // set c
19    public vo
20    {
21       commis
22    } // end
23
```

Fig. 10.7 | Commis

```
51
52    // return String representation of HourlyEmployee object
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    } // end method toString
59 } // end class HourlyEmployee
```

Fig. 10.6 | HourlyEmployee class derived from Employee. (Part 2 of 2.)

and social security number to the superclass Employee constructor (line 13) to initialize the private instance variables. In addition, method toString calls superclass method toString (line 56) to obtain the Employee-specific information (i.e., first name, last name and social security number)—this is another nice example of code reuse.

10.5.4 Creating Concrete Subclass CommissionEmployee

Class CommissionEmployee (Fig. 10.7) extends class Employee (line 4). The class includes a constructor (lines 10–16) that takes a first name, a last name, a social security number, a sales amount and a commission rate; *set* methods (lines 19–22 and 31–34) to assign new values to instance variables commissionRate and grossSales, respectively; *get* methods (lines 25–28 and 37–40) that retrieve the values of these instance variables; method earnings (lines 43–46) to calculate a CommissionEmployee's earnings; and method toString

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // five-argument constructor
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    } // end five-argument CommissionEmployee constructor
17
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22    } // end method setCommissionRate
23
```

Fig. 10.7 | CommissionEmployee class derived from Employee. (Part 1 of 2.)

```
24    // return commission rate
25    public double getCommissionRate()
26    {
27       return commissionRate;
28    } // end method getCommissionRate
29
30    // set gross sales amount
31    public void setGrossSales( double sales )
32    {
33       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34    } // end method setGrossSales
35
36    // return gross sales amount
37    public double getGrossSales()
38    {
39       return grossSales;
40    } // end method getGrossSales
41
42    // calculate earnings; override abstract method earnings in Employee
43    public double earnings()
44    {
45       return getCommissionRate() * getGrossSales();
46    } // end method earnings
47
48    // return String representation of CommissionEmployee object
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    } // end method toString
56 } // end class CommissionEmployee
```

Fig. 10.7 | `CommissionEmployee` class derived from `Employee`. (Part 2 of 2.)

(lines 49–55), which returns the employee's type, namely, `"commission employee: "` and employee-specific information. The `CommissionEmployee`'s constructor also passes the first name, last name and social security number to the `Employee` constructor (line 13) to initialize `Employee`'s `private` instance variables. Method `toString` calls superclass method `toString` (line 52) to obtain the `Employee`-specific information (i.e., first name, last name and social security number).

10.5.5 Creating Indirect Concrete Subclass BasePlusCommissionEmployee

Class `BasePlusCommissionEmployee` (Fig. 10.8) extends class `CommissionEmployee` (line 4) and therefore is an indirect subclass of class `Employee`. Class `BasePlusCommissionEmployee` has a constructor (lines 9–14) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes the first name, last name, social security number, sales amount and commission rate to the `CommissionEmployee` constructor (line 12) to initialize the inherited members. `BasePlusCommis-`

```
1  // Fig. 10.8:
2  // BasePlusCo
3
4  public class
5  {
6     private do
7
8     // six-arg
9     public Bas
10       String
11    {
12       super(
13       setBase
14    } // end s
15
16    // set bas
17    public voi
18    {
19       baseSal
20    } // end m
21
22    // return
23    public dou
24    {
25       return
26    } // end m
27
28    // calcula
29    public dou
30    {
31       return
32    } // end m
33
34    // return
35    public Str
36    {
37       return
38          "bas
39          "bas
40    } // end m
41 } // end clas
```

Fig. 10.8 | `BasePlu`

`sionEmployee` also
variable `baseSalary`
earnings (lines 29-
line 31 in method ea
culate the commissi
code reuse. `BasePlu`
representation of a
lowed by the `Stri`
method (another ex

```
1  // Fig. 10.8: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class extends CommissionEmployee.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
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 ); // validate and store base salary
14    } // end six-argument BasePlusCommissionEmployee constructor
15
16    // set base salary
17    public void setBaseSalary( double salary )
18    {
19       baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
20    } // end method setBaseSalary
21
22    // return base salary
23    public double getBaseSalary()
24    {
25       return baseSalary;
26    } // end method getBaseSalary
27
28    // calculate earnings; override method earnings in CommissionEmployee
29    public double earnings()
30    {
31       return getBaseSalary() + super.earnings();
32    } // end method earnings
33
34    // return String representation of BasePlusCommissionEmployee object
35    public String toString()
36    {
37       return String.format( "%s %s; %s: $%,.2f",
38          "base-salaried", super.toString(),
39          "base salary", getBaseSalary() );
40    } // end method toString
41 } // end class BasePlusCommissionEmployee
```

Fig. 10.8 | `BasePlusCommissionEmployee` class derived from `CommissionEmployee`.

`sionEmployee` also contains a *set* method (lines 17–20) to assign a new value to instance variable `baseSalary` and a *get* method (lines 23–26) to return `baseSalary`'s value. Method `earnings` (lines 29–32) calculates a `BasePlusCommissionEmployee`'s earnings. Note that line 31 in method `earnings` calls superclass `CommissionEmployee`'s `earnings` method to calculate the commission-based portion of the employee's earnings. This is a nice example of code reuse. `BasePlusCommissionEmployee`'s `toString` method (lines 35–40) creates a string representation of a `BasePlusCommissionEmployee` that contains `"base-salaried"`, followed by the `String` obtained by invoking superclass `CommissionEmployee`'s `toString` method (another example of code reuse), then the base salary. The result is a `String` begin-

ning with "base-salaried commission employee" followed by the rest of the BasePlusCommissionEmployee's information. Recall that CommissionEmployee's toString obtains the employee's first name, last name and social security number by invoking the toString method of its superclass (i.e., Employee)—yet another example of code reuse. Note that BasePlusCommissionEmployee's toString initiates a chain of method calls that span all three levels of the Employee hierarchy.

10.5.6 Demonstrating Polymorphic Processing, Operator instanceof and Downcasting

To test our Employee hierarchy, the application in Fig. 10.9 creates an object of each of the four concrete classes SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee. The program manipulates these objects, first via variables of each object's own type, then polymorphically, using an array of Employee variables. While processing the objects polymorphically, the program increases the base salary of each BasePlusCommissionEmployee by 10% (this, of course, requires determining the object's type at execution time). Finally, the program polymorphically determines and outputs the type of each object in the Employee array. Lines 9–18 create objects of each of the four concrete Employee subclasses. Lines 22–30 output the string representation and earnings of each of these objects. Note that each object's toString method is called implicitly by printf when the object is output as a String with the %s format specifier.

Line 33 declares employees and assigns it an array of four Employee variables. Line 36 assigns to element employees[0] the reference to a SalariedEmployee object. Line 37 assigns to element employees[1] the reference to an HourlyEmployee object. Line 38 assigns to element employees[2] the reference to a CommissionEmployee object. Line 39 assigns to element employee[3] the reference to a BasePlusCommissionEmployee object.

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String args[] )
7     {
8        // create subclass objects
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" );
```

Fig. 10.9 | Employee class hierarchy test program. (Part 1 of 3.)

```
21
22       System
23          sal
24       System
25          hou
26       System
27          com
28       System
29          bas
30          "ea
31
32       // cre
33       Employ
34
35       // ini
36       employ
37       employ
38       employ
39       employ
40
41       System
42
43       // gen
44       for (
45       {
46          Sys
47
48          //
49          if
50          {
51
52
53
54
55
56
57
58
59
60
61          } /
62
63          Sys
64
65       } // e
66
67       // get
68       for (
69          Sys
70
71    } // end
72 } // end cla
```

Fig. 10.9 | Employ