Common Programming Error 12.2

Failure to override a pure `virtual` function in a derived class makes that class abstract. Attempting to instantiate an object of an abstract class causes a compilation error.

Software Engineering Observation 12.10

An abstract class has at least one pure `virtual` function. An abstract class also can have data members and concrete functions (including constructors and destructors), which are subject to the normal rules of inheritance by derived classes.

12.5 Abstract Classes and Pure `virtual` Functions (cont.)

- Although we *cannot* instantiate objects of an abstract base class, we *can* use the abstract base class to declare *pointers* and *references* that can refer to objects of any *concrete* classes derived from the abstract class.
- Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

12.6 Case Study: Payroll System Using Polymorphism

- This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored throughout Section 11.3. We use an abstract class and polymorphism to perform payroll calculations based on the type of employee.

12.6 Case Study: Payroll System Using Polymorphism (cont.)

- We create an enhanced employee hierarchy to solve the following problem:
  - A company pays its employees weekly. The employees are of three types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, commission employees are paid a percentage of their sales and base-salary-plus-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants to implement a C++ program that performs its payroll calculations polymorphically-.
- We use abstract class `Employee` to represent the general concept of an employee.

12.6 Case Study: Payroll System Using Polymorphism (cont.)

- The UML class diagram in Fig. 12.7 shows the inheritance hierarchy for our polymorphic employee payroll application.
- The abstract class name `Employee` is italicized, as per the convention of the UML.
- Abstract base class `Employee` declares the "interface" to the hierarchy—that is, the set of member functions that a program can invoke on all `Employee` objects.
- Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so `private` data members `firstName`, `lastName` and `socialSecurityNumber` appear in abstract base class `Employee`.

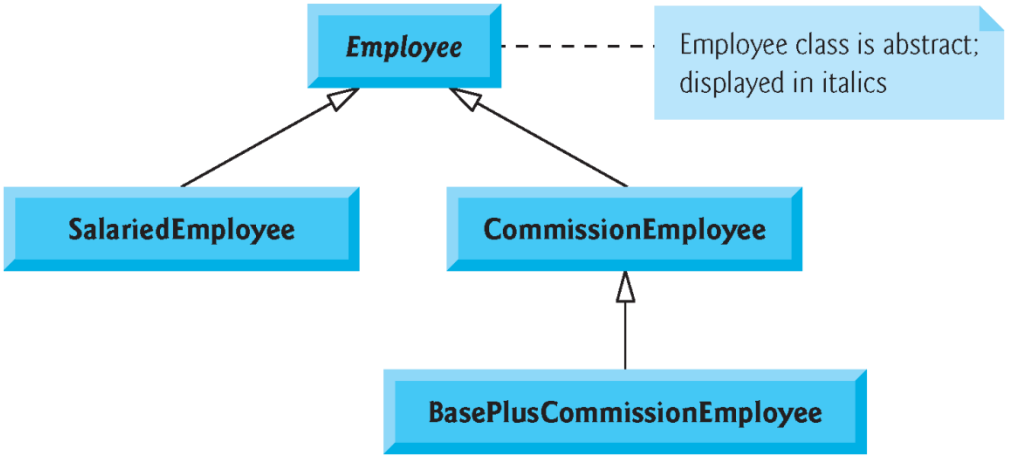

Fig. 12.7 | `Employee` hierarchy UML class diagram.

Software Engineering Observation 12.11

A derived class can inherit interface and/or implementation from a base class. Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy—each new derived class inherits one or more member functions that were defined in a base class, and the derived class uses the base-class definitions. Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy—a base class specifies one or more functions that should be defined for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s).

12.6.1 Creating Abstract Base Class `Employee`

- Class `Employee` (Figs. 12.9–12.10, discussed in further detail shortly) provides functions `earnings` and `print`, in addition to various *get* and *set* functions that manipulate `Employee`'s data members.
- An `earnings` function certainly applies generally to all employees, but each earnings calculation depends on the employee's class.
- So we declare `earnings` as pure `virtual` in base class `Employee` because a *default implementation does not make sense* for that function—there is not enough information to determine what amount `earnings` should return.
- Each derived class *overrides* `earnings` with an appropriate implementation.

12.6.1 Creating Abstract Base Class `Employee` (cont.)

- To calculate an employee's earnings, the program assigns the address of an employee's object to a base class `Employee` pointer, then invokes the `earnings` function on that object.
- We maintain a `vector` of `Employee` pointers, each of which points to an `Employee` object (of course, there cannot be `Employee` objects, because `Employee` is an abstract class—because of inheritance, however, all objects of all concrete derived classes of `Employee` may nevertheless be thought of as `Employee` objects).
- The program iterates through the `vector` and calls function `earnings` for each `Employee` object.
- C++ processes these function calls *polymorphically*.
- Including `earnings` as a pure `virtual` function in `Employee` forces every direct derived class of `Employee` that wishes to be a concrete class to override `earnings`.
- This enables the designer of the class hierarchy to demand that each derived class provide an appropriate pay calculation, if indeed that derived class is to be concrete.

12.6.1 Creating Abstract Base Class `Employee` (cont.)

- Function `print` in class `Employee` displays the first name, last name and social security number of the employee.
- As we'll see, each derived class of `Employee` overrides function `print` to output the employee's type (e.g., `"salaried employee:"`) followed by the rest of the employee's information.
- Function `print` could also call `earnings`, even though `print` is a pure-`virtual` function in class `Employee`.
- The diagram in Fig. 12.8 shows each of the five classes in the hierarchy down the left side and functions `earnings` and `print` across the top.

12.6.1 Creating Abstract Base Class `Employee` (cont.)

- For each class, the diagram shows the desired results of each function.
- Italic text represents where the values from a particular object are used in the `earnings` and `print` functions.
- Class `Employee` specifies "`= 0`" for function `earnings` to indicate that this is a pure `virtual` function.
- Each derived class overrides this function to provide an appropriate implementation.

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	*weeklySalary*	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*
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. 12.8 | Polymorphic interface for the `Employee` hierarchy classes.

12.6.1 Creating Abstract Base Class `Employee` (cont.)

`Employee` Class Header

- Let's consider class `Employee`'s header (Fig. 12.9).
- The `public` member functions include a constructor that takes the first name, last name and social security number as arguments (lines 11-12); a virtual destructor (line 13); *set* functions that set the first name, last name and social security number (lines 15, 18 and 21, respectively); *get* functions that return the first name, last name and social security number (lines 16, 19 and 22, respectively); pure `virtual` function `earnings` (line 25) and `virtual` function `print` (line 26).

12.6.1 Creating Abstract Base Class `Employee` (cont.)

`Employee` Class Member-Function Definitions

- Figure 12.10 contains the member-function definitions for class `Employee`.
- No implementation is provided for `virtual` function `earnings`.