chy as a class `Employee` in which each `Employee` *has a* different `CompensationModel` object. In this exercise, reimplement Exercise 9.16's `CompensationModel` class as an interface that provides a `public abstract` method `earnings` that receives no parameters and returns a `double`. Then create the following classes that implement interface `CompensationModel`:

a) `SalariedCompensationModel`—For `Employees` who are paid a fixed weekly salary, this class should contain a `weeklySalary` instance variable, and should implement method `earnings` to return the `weeklySalary`.

b) `HourlyCompensationModel`—For `Employees` who are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours per week, this class should contain `wage` and `hours` instance variables, and should implement method `earnings` based on the number of hours worked (see class `HourlyEmployee`'s `earnings` method in Fig. 10.6).

c) `CommissionCompensationModel`—For `Employees` who are paid by commission, this class should contain `grossSales` and `commissionRate` instance variables, and should implement method `earnings` to return `grossSales * commissionRate`.

d) `BasePlusCommissionCompensationModel`—For `Employees` who are paid a base salary and commission, this class should contain instance variables `grossSales`, `commissionRate` and `baseSalary` and should implement `earnings` to return `baseSalary + grossSales * commissionRate`.

In your test application, create `Employee` objects with each of the CompensationModels described above, then display each `Employee`'s earnings. Next, change each `Employee`'s `CompensationModel` dynamically and redisplay each `Employee`'s earnings.

10.18 *(Recommended Project: Implementing the `Payable` Interface)* Modify class `Employee` from Exercise 10.17 so that it implements the `Payable` interface of Fig. 10.11. Replace the `SalariedEmployee` objects in the application of Fig. 10.14 with the `Employee` objects from Exercise 10.17 and demonstrate processing the `Employee` and `Invoice` objects polymorphically.

Making a Difference

10.19 *(`CarbonFootprint` Interface: Polymorphism)* Using interfaces, as you learned in this chapter, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three small classes unrelated by inheritance—classes `Building`, `Car` and `Bicycle`. Give each class some unique appropriate attributes and behaviors that it does not have in common with other classes. Write an interface `CarbonFootprint` with a `getCarbonFootprint` method. Have each of your classes implement that interface, so that its `getCarbonFootprint` method calculates an appropriate carbon footprint for that class (check out a few websites that explain how to calculate carbon footprints). Write an application that creates objects of each of the three classes, places references to those objects in `ArrayList<CarbonFootprint>`, then iterates through the `ArrayList`, polymorphically invoking each object's `getCarbonFootprint` method. For each object, print some identifying information and the object's carbon footprint.