

Chapter 13

Object-Oriented Programming

Most professional programming is done in languages based on the object-oriented paradigm. C# is no exception and is in fact one of the languages with the best support for proper object-oriented programming. Throughout the previous chapters of this book, you have actually made frequent use of many object-oriented principles and techniques. For instance, every user interface element such as Button, TextBox, and Label are all objects. In this chapter, you will discover what object-oriented programming (OOP) is, how you can create and work with your own objects, and how to develop applications using OOP.

One of the primary reasons object-oriented programming is so popular in professional programming is that if you follow proper object-oriented principles, it leads you in the direction of a well-designed system. However, care still has to be taken in how this is achieved. The goal is to create a set of classes that can be reused in different contexts. For instance, you will see an Employee class throughout this chapter that could be used in a Payroll system or a Performance Evaluation system. If it is used for both purposes, once it has been put into production, it cannot be changed without considering the effect on both systems.

Topics

13.1 Introduction to Objects and Classes	13.6 Introduction to Inheritance
13.2 Classes vs. Objects	13.7 Implementing Inheritance
13.3 Information Hiding (Encapsulation)	13.8 Using Subclasses and Superclasses
13.4 Properties	13.9 Overriding Methods
13.5 Calling Methods (Sending Messages to Objects)	13.10 Polymorphism

13.1 Introduction to Objects and Classes

Object-oriented programmers generally distinguish between the problem domain and the application domain. The problem domain involves the parts of the real world that the computer system is working with and solving problems for. For instance, the problem domain for a payroll system would contain the employees of the company, the actual hours worked by those employees, and all the rules governing how salaries, taxes, and other deductions are calculated and paid out. The application domain, on the other hand, is the actual payroll computer system and its users. The users will work with representations of the problem domain (real world) in order to solve the problems the system is intended to solve.

With object-oriented programming, we start by creating representations of the problem domain inside the application. The problem domain typically contains multiple entities like employee and payroll. The instances of entities we want to keep track of in the problem domain are represented in the application

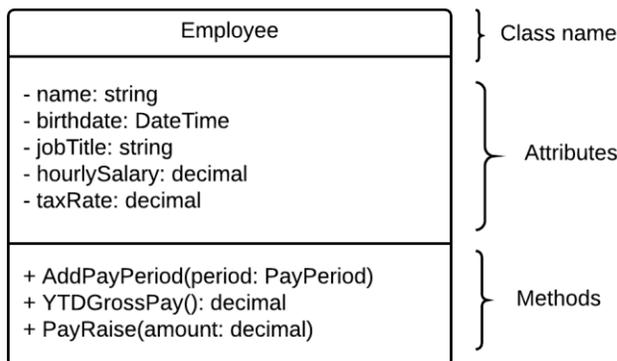
domain by *objects*. So, in the payroll system, each employee becomes an object, every pay period becomes an object, every pay check becomes an object, and so on. The set of objects that represent the instances of an entity is described in computer code with a construct called a *class*. A class is like a blueprint that can be used as a template to create many instances (objects) that have the same properties. In this way, a class represents an entity of the real-world problem addressed by the application. For example, a payroll application for an organization may use an Employee class to represent employees of the organization.

13.2 Classes vs. Objects

With object-oriented programming, you start by creating classes that represent real-world entities. A class consists of code that describes a group of data items that represent the attributes of the entity and methods that represent the behavior of the entity. The name, birthdate, and address of the employee are examples of the data items of the Employee class. The data items in a class are accessed using an interface of publicly available methods and properties of the class. Behaviors are the activities or functions of an entity. Updating the hourly rate for an employee would be a behavior of the employee entity, which may be implemented by a method called PayRaise within the Employee class.

Figure 13-1 shows the Employee class in Unified Modeling Language (UML) notation. In UML, a class is represented by a box divided into three sections. The top portion contains the class name, the middle portion contains the attributes or properties, and the lower portion contains the methods. The plusses and minuses signify whether the element is public or private, respectively. Public elements make up the public interface that other classes in the system can access, whereas private elements can only be accessed from within the class itself. The notation for the methods is similar to C# but also slightly different. The name of the method is given first, followed by parentheses that list the parameters that the method accepts as well as the data type for each parameter. If the method returns a value, the data type for the return value is given after the parameters. In this example, the PayRaiseAmount method is public and takes a single parameter of the type decimal. It doesn't return anything.

Figure 13-1: Employee class in UML notation



The Employee class describes the attributes and behavior of the employees of an organization. To represent an individual employee like John Smith, an application creates an instance of the Employee class in memory, called an object. So, the object is an *instantiation* of the class.

The class is an abstraction of the real-world entity written in computer code; thus, the same Employee class can be used to create multiple objects, each object representing a different employee. The class is often compared to the blueprint of a house, and the objects compared to multiple houses that are built from the same blueprint.

Objects in C# are characterized by three general concepts:

- **Identity:** Just like every employee is distinct from every other employee, so too is every object in the computer system distinct from every other object. Once an object has been created, we can distinguish it from all other objects in the system. This is similar to the concept of a primary key in a database, but object-oriented systems automatically implement an identity mechanism.
- **State:** The state of an object is the set of values of the attributes that we care about regarding that object. For employees, we would likely care about things like their name, birthdate, address, job title, and pay rate, whereas we are not likely to be concerned about their hair color. Each object has specific values for the things we care about. So, we might have two employees with these two states:

Table 13-1: Examples of objects

Attributes	Employee 1	Employee 2
<i>Name</i>	John Smith	Rebecca Jones
<i>Birthdate</i>	12/10/1993	10/5/1994
<i>Address</i>	200 Main St	100 Elm St
<i>Job Title</i>	Network Engineer	Software Developer
<i>Hourly Pay</i>	\$35	\$45

It's important to realize that the state of an object changes over time. In fact, anytime the value of an attribute changes, the state of the object has changed. So, for instance, if Rebecca gets a pay raise to \$47 per hour, the state of the Employee 2 object has changed.

- **Behavior:** Each object has specific behavior that is also modeled in the system. In the problem domain, we might have employees punch in for work, punch out, get their salary paid out, get a pay raise, etc. In the application domain, behavior is implemented as methods that can be called on an object. The method code specifies what action happens when the method is called on a particular object.

The concepts of class and objects are often confused and described in overlapping terms, but they are two distinct concepts that are important to keep separated. Classes are described in code and are used as the blueprints to instantiate (create) the objects. Each object in an application represents an instance of a real-world entity. There would only be one Employee class, but many Employee objects (one for each actual employee in the organization).

Review Questions

- 13.1 Identify several objects in your world from the following classes
- Book
 - Car
 - Account
 - Student
 - Professor
- 13.2 For each of the classes above, identify a few attributes and behaviors that might be relevant to represent in an information system.
- 13.3 In your own words, describe the difference between class and object.

13.3 Information Hiding (Encapsulation)

One of the defining principles of object-oriented programming is that of Information Hiding (sometimes also referred to as encapsulation). The idea is that the way the data is presented by an object to other parts of the system is independent of how it is actually stored in the object.

This distinction provides several advantages. First, it allows for a simple and consistent internal representation of data in an object. For example, the total time worked by an employee could be stored in minutes, but it could be presented in the public interface by a method that returns fractional hours. Second, it protects the state of the object from being changed in inappropriate ways. For example, if the time worked by an employee is really represented by successively punching in and out, it would be inappropriate to be able to change the total time worked directly; it should only be changed through the transactions. Lastly, it also allows for restricting how the data inside an object can be accessed. Some attributes of an object should not be changed from outside of that object. As an example, consider a pay rate for an employee that must fall within certain bounds. If the pay rate could be changed directly, it could not be guaranteed to stay within its bounds.

To achieve this, each object provides a private implementation of data and a public interface, and only the public interface is available to other parts of the system. The implementation is thus “hidden” or “encapsulated” inside the object. In Figure 13-1, the attributes and methods marked with a plus represent the public interface, whereas the ones with a minus are private. A UML class diagram might omit some private implementations, and as we discuss later, attributes are implemented in C# using private fields that are exposed through public **properties** and methods.

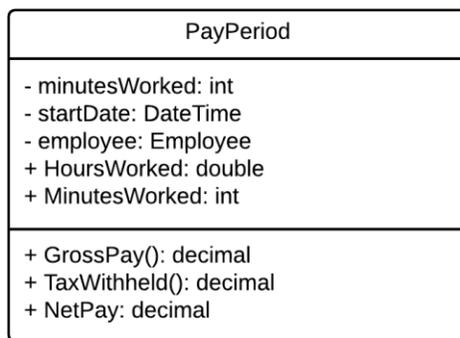
As an example, consider the Employee object described above. We have specified that it has a Name attribute. How should that be actually stored in the object? A simple approach could be to store it in a single string. However, you could also split it in two and store first and last name separately. In that case, the attribute that provides the full name would be responsible for combining the first and last name and presenting it to its clients in that form. We could also do the opposite and define public access to both first and last name. If the name was stored internally in a single string, the first and last name attributes would be responsible for extracting the proper string and returning. Similarly, for hourly pay, which is defined as a whole number of dollars, we could store this as a decimal and provide public access as a rounded value as an integer.

When you design the system and decide on how to represent the data inside the objects, some representations are clearly better than others, so you have to carefully design both the public interface and

the internal representation. It is much easier to combine a first and last name string into a full name than to attempt to create rules for extracting a first and last name from a full name.

As another example, consider the `PayPeriod` class shown in Figure 13-2. Each `PayPeriod` object contains data that allows for calculating the salary for an employee during that period. One of the items stored is the time that the employee worked during the pay period. But what is the best way to store this value? As whole minutes, fractional hours, or a combination of hours and minutes? This is obviously an important decision, but for the client classes that use the `PayPeriod`, it is largely irrelevant, as the programmer of the `PayPeriod` class could just provide methods for retrieving the time in all three formats and then have those methods do conversions as necessary.

Figure 13-2: `PayPeriod` class



There are several advantages to encapsulating the implementation like this:

- It allows for conversion between different formats or units—for example, between hours and minutes.
- It allows for having read-only and write-only attributes. There are many situations where it is useful to restrict the ability to change the attribute of a class directly. For instance, in an `Account` class you might have an attribute for the `Balance` on the account, but in most systems, the balance would only be changed by performing transactions on the account, and not by setting it directly. So, the `Balance` is better represented by a read-only attribute. Write-only attributes are rare in practice but would allow for the value to be changed but not read directly.
- The software becomes easier to maintain. By having a single internal representation and controlled access, there is only one place to make changes internally, and conversions can be done in a single spot as well.
- You will have fewer mistakes in the code due to unintended changes of values in an object. When access to the stored values in an object is properly protected, it is less likely to have unintended changes or end up with inconsistent values for an object.
- You can change the implementation of a class without affecting the public interface.

Review Questions

- 13.4 Imagine a Thermostat class that allows for a temperature to be set to specific values within minimum and maximum bounds. Discuss how encapsulation can be helpful for this class, as well as what problems might arise if data in the class were not encapsulated.
- 13.5 Draw a UML class diagram for the Thermostat class in the previous question.

Tutorial 1: Create an Employee Class

In this tutorial, you will create your first class and several objects from this class to be used in a payroll system. The class is the Employee class that was mentioned above. In the context of this simplified system, an employee will have the following attributes:

- Name—of type string
- Birthdate—of type DateTime
- Hourly salary—of type decimal
- Job title—of type string
- Tax rate—of type double (must be between 0 and 1)

Employee objects will have the following behavior:

- Given a number of hours worked during a pay period, calculate the gross salary, the tax amount, and the net amount of money to be paid out.

Step 1-1: Start Visual Studio and create a new Windows Forms Application project called Payroll.

Step 1-2: Add the Employee class.

Right click the project in the Solution Explorer and select Add > Class...
Name the class Employee.cs and click Add.

Fields

When implementing classes, fields are variables at the class level that represent the attributes for the class and allow for storing the values of the attributes (the state) of an object. Fields should always be marked private. If you do not add an access modifier, C# will default to the most restricted access you could declare for that member. This would be private for fields, but it is recommended to still add the private key word to signal your intent clearly.

Step 1-3: Add fields to the Employee class.

Add the code shown in Figure 13-3 to the Employee class.

Figure 13-3: Fields of Employee class

```
7 namespace Payroll
8 {
9     class Employee
10    {
11        private string name;
12        private DateTime birthday;
13        private decimal hourlySalary;
14        private string jobTitle;
15        private decimal taxRate;
16    }
17 }
```

Each of the fields declared here helps describe an Employee. Notice how they correspond to the attributes shown in Table 13-1. When you create different objects from the Employee class, each object will have a different set of values for the fields because each object represents a different employee.

Each field is declared using the private key word, which is a way to express that the variable cannot be directly accessed from outside the class and that details of how the data is stored is to be kept internal to the object. This will allow you to change the implementation without affecting the external usage of the class.

The Constructor

The constructor is a special method that is called when creating an object from a class. The constructor has two purposes: Create the object and initialize the fields. Creating the object happens behind the scenes by the runtime environment when your program is running; you don't have to do anything special in the constructor code. However, you will have to write the code to initialize the fields, which you will see in the next step.

The constructor takes parameters that can be used to initialize the fields. In this case, there are parameters for each of the fields except the tax rate, which is set to a default value of 0.25 (all employees pay 25% tax by default).

In C#, the constructor follows a specific pattern:

- Constructors are given the same name as the class.
- Constructors do not return anything (not even void).
- Each field is initialized in the constructor. If you do not explicitly initialize a field, it will get a default value, depending on its data type (numbers become zero, strings become null, Booleans become false, etc).

Step 1-4: Add the constructor to the Employee class.

Add the following code after the declaration of the fields, before the closing brace of the Employee class.

Figure 13-4: Employee constructor

```

16
17 public Employee(string name, DateTime birthday, decimal hourlySalary, string jobTitle)
18 {
19     this.name = name;
20     this.birthday = birthday;
21     this.hourlySalary = hourlySalary;
22     this.jobTitle = jobTitle;
23     taxRate = 0.25m;
24 }
25

```

The key word “this” is used to refer to the current object instance, and allows you to distinguish between fields and parameter values, as in line 19:

```

    this.name = name;

```

At first blush, this line looks a little strange, but it is actually quite simple: `this.name` refers to the field name (line 11 in Figure 13-3), and the name on the right side of the equal sign refers to the parameter name. You can see this in Visual Studio by placing the cursor inside each of the two occurrences of `name`. When you click on the one on the right, the parameter is highlighted, and when you click on `this.name`, the field is highlighted. It isn’t a requirement that the constructor parameters match the name of the fields, but it is fairly common that they do.

The ToString method

It is often useful to have an object be able to provide a brief one-line description of itself. This is usually done by implementing a method called `ToString`. There are technical reasons for the method to be called `ToString` that will be covered later, when we discuss the concept of *inheritance*. For now, just add this method to the `Employee` class.

Step 1-5: Add the `ToString` method to the `Employee` class.

Figure 13-5: `ToString` method

```

26 public override string ToString()
27 {
28     return string.Format("Name: {0}, Birthday: {1:d}, Hourly Salary: {2:c}, Job Title: {3}",
29         name, birthday, hourlySalary, jobTitle);
30 }

```

As you can see, the method just returns a string that includes most of the fields. The meaning of the `override` key word in line 26 will also be discussed later during the coverage of *inheritance*.

Step 1-6: Create the user interface.

Rename `Form1.cs` in Solution Explorer to `EmployeeForm.cs` (and rename in the code as well when you’re asked). Change the `Text` property of the form to `Employees`.

In order to demonstrate how objects are created, add a label named `lblEmployee` to the Form in the project.

Double click the background of the form (not the label) to switch to the code view of the form. It should look like Figure 13-6.

Figure 13-6: Initial code in the EmployeeForm class

```

11 namespace Payroll
12 {
13     public partial class EmployeeForm : Form
14     {
15         public EmployeeForm()
16         {
17             InitializeComponent();
18         }
19
20         private void EmployeeForm_Load(object sender, EventArgs e)
21         {
22
23         }
24     }
25 }

```

As you can see from line 13, the Form is in fact a class, called EmployeeForm. This means that when the program is running and the Form is being displayed, an EmployeeForm object is created. This class currently has two methods:

1. A constructor, specified in lines 15–18, that takes no parameters and calls the method `InitializeComponent`, which is created by Visual Studio to create all the user interface components in the form. If you want to see this auto-generated code, you can right click on the method name and select `Go To Definition`.
2. A method called `EmployeeForm_Load`, specified in lines 20–23, that takes two arguments. The `EmployeeForm_Load` method is called automatically by the system when the form is loaded. There are a number of other event handler methods that you can add to your form and insert code into to execute code at certain times in the life cycle of a form.

Creating an Object

When creating an object, you follow a specific pattern:

```
ClassName variableName = new ClassName();
```

This line declares a variable on the left side of the equal sign and on the right side calls the constructor for the class. The right side generates an object that is then assigned to the variable on the left side.

The key here is the use of the `new` key word in front of the call to the constructor. The `new` key word is used to indicate that an object is created from the specified class. Let's create our first Employee object.

Step 1-7: Create objects and display ToString.

Add the following code to the `EmployeeForm_Load` event handler:

```

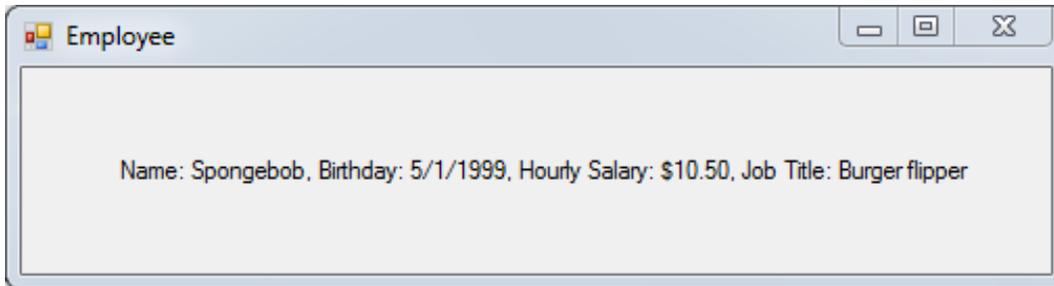
DateTime birthday = new DateTime(1999, 5, 1);
Employee spongebob = new Employee("Spongebob", birthday, 10.5m, "Burger flipper");
lblEmployee.Text = spongebob.ToString();

```

This code actually creates two objects. First, a `DateTime` object is created for May 1, 1999. This is assigned to the variable `birthday`. Then, an `Employee` object is created using the `birthday` object and several other pieces of data. This is assigned to the `spongebob` variable. Finally, the `ToString` method is called on the `spongebob` object, and the result is assigned to the `Text` property of the label you added to the user interface above.

Run the program and you should see the description of the `Employee` object displayed on the form, as shown in Figure 13-7.

Figure 13-7: First object created



To make sure you understand how the object is created and displayed, add a `Breakpoint` on the second line of code in the `EmployeeForm_Load` event handler. Once the debugger stops on that line, examine the variables available at that point.

Next, step into the `Employee` constructor, and see how the values are initialized in the object. Once you return to the `Form` object, reexamine the variables. You should now see that the `spongebob` variable has values associated with each of the fields.

Continue stepping into the `ToString` method, and notice that you are back in the `Employee` class looking at the `spongebob` object.

It's time to practice! Do Step 1-8 and Exercise 13-1 at the end of the chapter.

Step 1-8: Add a second label and create a new `Employee` object with different values and show the new object on the new label.

13.4 Properties

So far, you have seen fields and methods in a class. The fields contain the state of the object, and the methods implement the object's behavior. Because of encapsulation, the fields are always kept private so as not to allow other parts of the system to access the fields directly. However, in many cases we want to be able to expose the fields and be able to read and change them in a controlled manner. For instance, it is easy to imagine that you could create a form with controls for all the fields for the `Employee` class and allow a user to enter values into the controls and assign those values to the fields. But because the fields are private, this would not be possible. You can try right now in the program you created previously to see if you can assign a different value to one of the variables in the `spongebob` object after it has been created by adding this line after creating the object:

```
spongebob.name = "Spongebob Squarepants";
```

You will get a syntax error from trying this, because `name` is declared as private in the `Employee` class, so go ahead and comment out that line of code.

So, how can we allow the fields to remain private but still change their values? The C# language to a large extent was created by Microsoft as an improved version of Java. The best way to understand why C# has a concept of properties is to go back to the Java language.

In Java we would create public “getters” and “setters” to retrieve and change the field values. For the field, name, these would look like this:

```
public string getName()
{
    return this.name;    //return the value of the name field
}
public void setName(string value)
{
    this.name = value;    //assign a value to the name field
}
```

If a programmer wanted to not allow for changing a field, he/she would simply not write the set method. By keeping the fields private, access to them could be controlled through the getters and setters. For instance, if it was necessary to do any kind of conversion, like splitting the name into last and first names, this could be done in the getters and setters.

If you implemented these methods, you could change the name with this line of code:

```
Spongebob.setName("Spongebob Squarepants");
```

However, one problem with this approach is that getters and setters are not a part of the Java language; they are merely a convention that most Java programmers follow. So, the C# language designers decided to introduce the concept of a **Property** that takes the place of the getters and setters in Java. Here is a property that provides the same functionality as the getters and setters above:

```
public string Name
{
    get
    {
        return this.name; //return the value of the name field
    }
    set
    {
        this.name = value; //assign a value to the name field
    }
}
```

This is a single construct called Name that includes a get section and a set section. The get simply returns the value in the named field. The set uses the reserved key word *value* to provide the value to be stored in the field. Just like with getters and setters in Java, you can add any valid C# code to the properties. You're not restricted to just assigning and returning values from fields. You could do conversions or have properties whose values are calculated and do not have an associated field.

Once the property has been added, you can change the name in this way:

```
spongebob.Name = "Spongebob Squarepants";
```

This is very similar to your first attempt at changing the name, but because the property is declared public, it will actually work.

How do you retrieve the name? You have seen many examples throughout this book of using properties, as they are very common with user interface controls and other parts of the .Net framework, so a single additional example here should suffice. Imagine a form that has a label called lblName and a TextBox named txtName:

```
lblName.Text = spongebob.Name;
spongebob.Name = txtName.Text;
```

In the first statement, the Name property of the spongebob object retrieves the value of the name field using the get section of the property. The value is assigned to the Text property of the Label.

In the second statement, the content of the textbox is retrieved using the Text property of the txtName object and assigned to the Name property of the spongebob object. When a value is assigned to a property, the set section of the property is used to assign the value to the field through the value keyword in the set section of the property. So, the content of txtName is assigned to the name field of spongebob object.

The naming convention for properties is to change the first letter of the corresponding field to uppercase. If you wanted to have a read-only property, you would simply leave out the set section.

Automatic Properties

One final note about properties is that very often there is a lot of boilerplate code when creating a class:

- A private field
- A public property that simply gets and sets the values for the fields

When this is the case, the field and property code can be written using a simplified version, called an Automatic Property. Here's the Name property from earlier using the automatic Property:

```
public string Name { get; set; }
```

This one line of code replaces the following private field and the property code:

```
private string name;
public string Name
{
    get
    {
        return this.name;
    }
    set
    {
        this.name = value;
    }
}
```

As you can see, Automatic properties are much shorter and easier to read. The usage remains the same as you saw above, except the field can no longer be accessed directly inside the property or in the rest of the class.

With an automatic property, the compiler will automatically create a backing field of the correct type and implement the standard get and set code you saw for the fully implemented property. So, the only difference is that it is much shorter to write and to read. When you use automatic properties, you have to

remember that you cannot implement any conversion or other code in the get and set sections. You must also supply both get and set, so you cannot create automatic read-only or write-only properties.

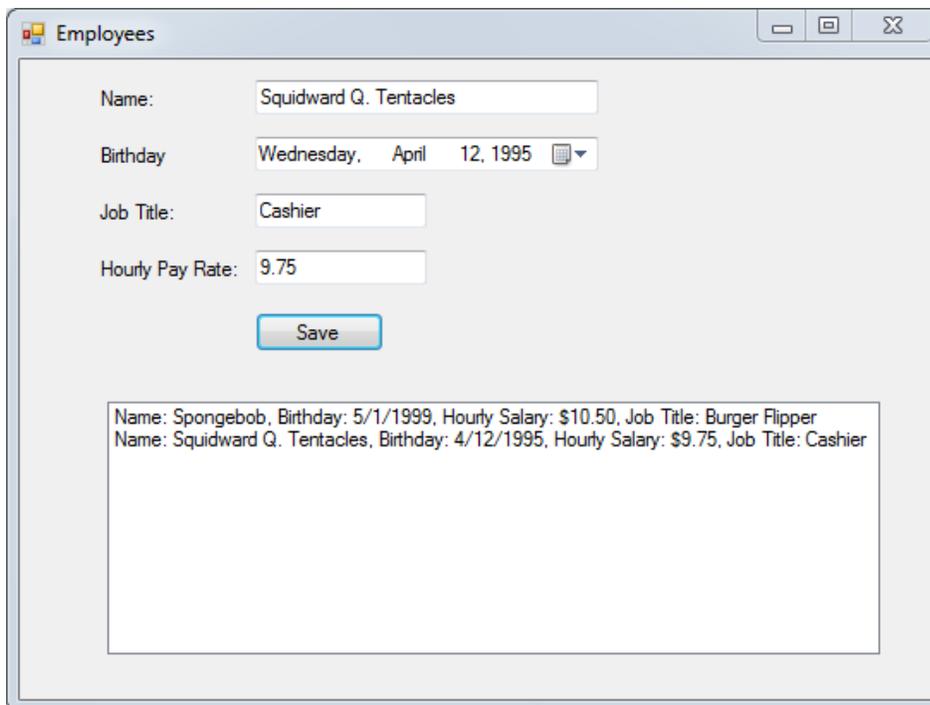
Review Questions

- 13.6 What are the advantages and disadvantages of automatic properties?
- 13.7 Why should fields always be marked as private?
- 13.8 Why did the C# language designers choose to introduce properties instead of getters and setters like in Java?

Tutorial 2: User Interface and Properties

In this tutorial you will learn how to store multiple objects in a `ListBox` and retrieve them. This will help you understand how to work with references to objects. By the end of this tutorial, you will have a form where the user can enter information on an employee and click a button to create an `Employee` object. Multiple `Employee` objects will be created and displayed in a `ListBox` where the user can select one to be modified. Figure 13-8 shows the final user interface for Tutorial 2.

Figure 13-8: Final user interface for Tutorial 2



The screenshot shows a Windows application window titled "Employees". The window contains a form with the following fields and controls:

- Name:** A text box containing "Squidward Q. Tentacles".
- Birthday:** A date picker showing "Wednesday, April 12, 1995".
- Job Title:** A text box containing "Cashier".
- Hourly Pay Rate:** A text box containing "9.75".
- Save:** A button.
- ListBox:** A list box containing two entries:
 - Name: Spongebob, Birthday: 5/1/1999, Hourly Salary: \$10.50, Job Title: Burger Flipper
 - Name: Squidward Q. Tentacles, Birthday: 4/12/1995, Hourly Salary: \$9.75, Job Title: Cashier

Step 2-1: Create a user interface form.

Continue working with the project from Tutorial 1. Move the Label you added earlier to the bottom of the Employee form and add controls to the `EmployeeForm` in the project so it looks like Figure 13-9.

Figure 13-9: Employees form

Name the controls on the right as follows (from top to bottom): txtName, dtpBirthday, txtJobTitle, txtPayRate, and btnSave.

Step 2-2: Code the button to create an object and display it in the label:

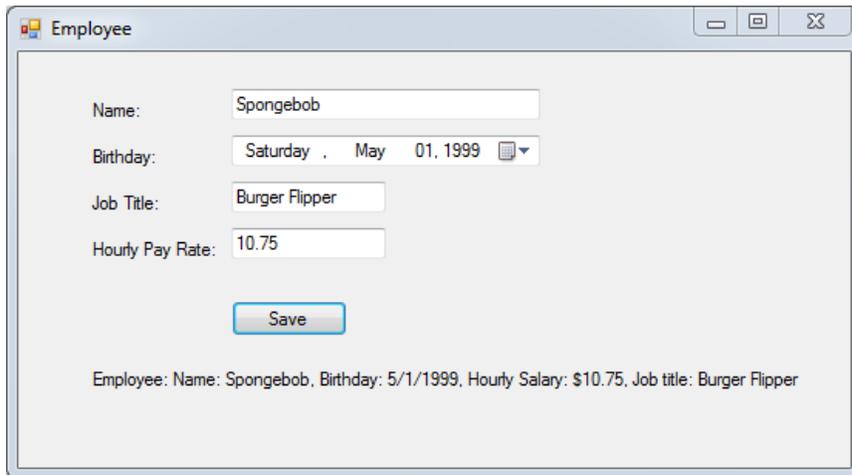
Double click the Save button to go to the code for the form.

Comment out the code you added to the Form_Load method and add the following code to the btnSave_Click method:

```
private void btnSave_Click(object sender, EventArgs e)
{
    string name = txtName.Text;
    DateTime birthday = dtpBirthday.Value;
    string title = txtJobTitle.Text;
    decimal hourlyRate = Decimal.Parse(txtPayRate.Text);
    Employee employee = new Employee(name, birthday, hourlyRate, title);
    lblEmployee.Text = "Employee: " + employee.ToString();
}
```

The code is fairly straightforward. First, all the values are read from the user interface; then an Employee object is created using the new key word. Finally, the lblEmployee's Text property is set to the result of calling the ToString method on the employee object.

Run the program and add values to the form fields. Click the Save button, and notice that the label at the bottom of the form is updated with information about the object, as shown in Figure 13-10.

Figure 13-10: Output from creating a single object

Step 2-3: To help save multiple objects, add a `ListBox` and add objects to the `ListBox` with the `ToString` method.

Remove the `lblEmployee` label from the form and replace it with a `ListBox` named `lstEmployees`.

Now instead of adding each object to the label and losing them, we will add them to the `ListBox` and explore some ways to interact with them again later.

Remove the last line of code in the `btnSave_Click` method and replace it with this line of code:

```
lstEmployees.Items.Add(employee);
```

This line will add the `employee` object to the `Items` collection in the `ListBox`. Previously, when you have worked with `ListBoxes`, you added string objects to the `Items` collection. In this case, you have added an entire `Employee` object with several pieces of data included. However, when the `ListBox` is being displayed, it will automatically call the `ToString` method on each of the objects in its `Items` collection, and will then display the return value (string) from calling the method.

Run the code again, and you should see that the `employee` objects show up one after the next in the `ListBox` each time you click `Save`, as shown in Figure 13-8.

Step 2-4: To make it possible to change the values of the object, add properties to the `Employee` class.

So far, you can only add objects to the list, and there is no way to edit an existing object. To fix this, you will have to allow the user to select an item in the `ListBox` and then display the values of the fields for the object in the form. The behavior of the `Save` button must be changed to recognize whether an `employee` has been selected and then modify the selected object rather than create a new one. However, all the fields are private in the `Employee` class, so the first thing you need to do is to expose them as properties, so you can read and modify them as needed.

Comment out all the fields in the Employee class except taxRate and replace them with the following code:

```
//private string name;
public string Name { get; set; }
//private DateTime birthday;
public DateTime Birthday { get; set; }
//private decimal hourlySalary;
public decimal HourlySalary { get; set; }
//private string jobTitle;
public string JobTitle { get; set; }
private decimal taxRate;
public decimal TaxRate { get { return taxRate; } }
```

Now the code in the constructor and ToString is broken because the fields no longer exist. The properties are declared public and as such are always named with a starting uppercase letter. The fields were private and, as such, named with a lowercase initial letter. Because C# is case sensitive, the two are not interchangeable.

Because the code uses automatic property for all attributes, except taxRate, there is no explicit declaration of fields for those attributes. The taxRate field was left unchanged and a property added. We didn't use an automatic property for taxRate because we wanted to make it a read-only property so that the tax rate is not allowed to be changed.

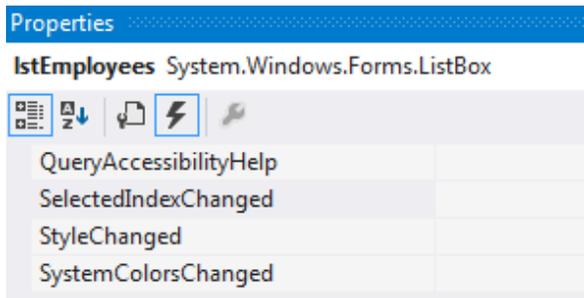
Step 2-5: Fix the problems by changing the lowercase initial letter to uppercase for each of the errors in the constructor and ToString method (look for the squiggly red lines). See Figure 13-11 for the updated code. Run the form to test the code.

Figure 13-11: Updated constructor and ToString method in Employee

```
20 public Employee(string name, DateTime birthday, decimal hourlySalary, string jobTitle)
21 {
22     this.Name = name;
23     this.Birthday = birthday;
24     this.HourlySalary = hourlySalary;
25     this.JobTitle = jobTitle;
26     taxRate = 0.25m;
27 }
28
29 public override string ToString()
30 {
31     return string.Format("Name: {0}, Birthday: {1:d}, Hourly Salary: {2:c}, Job title: {3}",
32         Name, Birthday, HourlySalary, JobTitle);
33 }
```

Step 2-6: Recall the object from the ListBox and display in the user interface form.

Switch to the design view of the form and select the ListBox. Switch the Properties window to events by clicking the lightning bolt. Then find the SelectedIndexChanged event and double click in the field next to it (see Figure 13-12).

Figure 13-12: SelectedIndexChanged event for the ListBox

In the generated event handler, enter the following code:

```
private void lstEmployees_SelectedIndexChanged(object sender, EventArgs e)
{
    Employee employee = (Employee)lstEmployees.SelectedItem;
    txtName.Text = employee.Name;
    txtPayRate.Text = employee.HourlySalary.ToString();
    dtpBirthday.Value = employee.Birthday;
    txtJobTitle.Text = employee.JobTitle;
}
```

The first statement in the event handler will get the item that the user selected in the ListBox (SelectedItem property). This item, which is of type object, is then cast to an Employee and assigned to the employee variable. The last four lines then change the user interface controls based on the values in the object. Notice how the properties in the Employee class are used to extract the values from the object.

Items in the ListBox are stored in a collection called Items. When you add an object to that collection, the ListBox will display it by calling the ToString method to display it in the ListBox. Since you have been storing Employee objects, you can get that entire object back and interact with it.

A common mistake when working with ListBoxes is to add a string describing the object, such as the name of the employee. Then, to retrieve it, you would have to get the index it was stored at to be used to find the object in some other collection. However, this approach is error prone because it assumes that the ListBox and the collection are kept in sync, with no objects added or deleted or any sorting applied to one but not the other. By saving the entire object to the ListBox, it doesn't matter what happens to the other collection.

Step 2-7: Save object values.

Once the object is displayed in the form fields, we need to be able to save any changes the user makes back into the object rather than have it save as a new object when the Save button is clicked. To do this, you will use the properties again to save the values. However, first you have to find the right object to save into. In the previous step, the employee variable only has scope within the lstEmployees_SelectedIndexChanged method, so it cannot be accessed once that method finishes executing. However, the object is still selected in the ListBox, so you can check to see if an item is selected and then save the values into the selected item's object. Otherwise, if no object is selected, create a new object and add to the list.

The first step is to recall the object and update the properties. Revise the btnSave_Click method, as shown in Figure 13-13.

Figure 13-13: Making changes to object properties

```

37     private void btnSave_Click(object sender, EventArgs e)
38     {
39         //Grab values from the form.
40         string name = txtName.Text;
41         DateTime birthday = dtpBirthday.Value;
42         string title = txtJobTitle.Text;
43         decimal hourlyRate = Decimal.Parse(txtPayRate.Text);
44         Employee employee;
45         if (lstEmployees.SelectedIndex >= 0)
46         {
47             //If an employee is selected in the ListBox, get reference
48             //to that employee and update properties from the form.
49             employee = (Employee)lstEmployees.SelectedItem;
50             employee.Name = name;
51             employee.Birthday = birthday;
52             employee.JobTitle = title;
53             employee.HourlySalary = hourlyRate;
54         }
55         else
56         {
57             //If no employee is selected, create a new object based on
58             //values in the form
59             employee = new Employee(name, birthday, hourlyRate, title);
60             lstEmployees.Items.Add(employee); //Add new object to ListBox
61         }
62     }

```

Line 44 declares a variable of type `Employee`, which is populated in one of two ways—either by the employee selected in the `ListBox` (line 49) or as a new object (line 59).

The `if` statement in Line 45 checks whether an item is selected in the `ListBox`. If selected, the object is retrieved (line 49) and the object values are updated with the updated data (lines 50–53); if not, a new `Employee` object is created in line 59 and added as a new item to the `ListBox` in line 60.

If you run the program after making the changes in the above step, the `ListBox` doesn't behave as it should. When clicking `Save` after updating an existing employee, the corresponding employee information displayed in the `ListBox` doesn't update. So if you select an employee that has been previously updated, the `ListBox` and the user interface fields won't show the same thing. There are a few ways to fix this, but the simplest is to keep a `List` (a collection) of the `Employees` as a field in the `EmployeeForm` class. Then every time a new `Employee` object is created, it is added to the `List`, and `Employee` objects in the `List` are updated without adding new objects. Every time the `List` is updated, you can clear the items out of the `ListBox` and refresh with the items in the `List`.

Step 2-8: Refresh the `ListBox`.

To refresh the `ListBox`, first create a private field called `employees` at the top of the `EmployeeForm` class and instantiate it in the `EmployeeForm` constructor as shown in line 20 of Figure 13-14.

Figure 13-14: List for storing Employee objects

```
13 public partial class EmployeeForm : Form
14 {
15     private List<Employee> employees;
16
17     public EmployeeForm()
18     {
19         InitializeComponent();
20         employees = new List<Employee>();
21     }
```

Line 15 declares a List that can hold Employee objects, and line 20 creates the List object.

Next, update the code in btnSave_Click as shown in Figure 13-15.

Figure 13-15: Using a List to store Employee objects

```
37 private void btnSave_Click(object sender, EventArgs e)
38 {
39     //Grab values from the form.
40     string name = txtName.Text;
41     DateTime birthday = dtpBirthday.Value;
42     string title = txtJobTitle.Text;
43     decimal hourlyRate = Decimal.Parse(txtPayRate.Text);
44     Employee employee;
45     if (lstEmployees.SelectedIndex >= 0)
46     {
47         //If an employee is selected in the ListBox, get reference
48         //to that employee and update properties from the form.
49         employee = (Employee)lstEmployees.SelectedItem;
50         employee.Name = name;
51         employee.Birthday = birthday;
52         employee.JobTitle = title;
53         employee.HourlySalary = hourlyRate;
54     }
55     else
56     {
57         //If no employee is selected, create a new object based on
58         //values in the form
59         employee = new Employee(name, birthday, hourlyRate, title);
60         //lstEmployees.Items.Add(employee); //Add new object to ListBox
61         employees.Add(employee); //Add new object to the collection
62     }
63     //Clear all items from the ListBox
64     lstEmployees.Items.Clear();
65     //Add all items from the List to the ListBox
66     lstEmployees.Items.AddRange(employees.ToArray());
67 }
```

Line 60 has been replaced by line 61, which adds the newly created Employee object to the List. Note that an object is added to the List only if it is a newly created object.

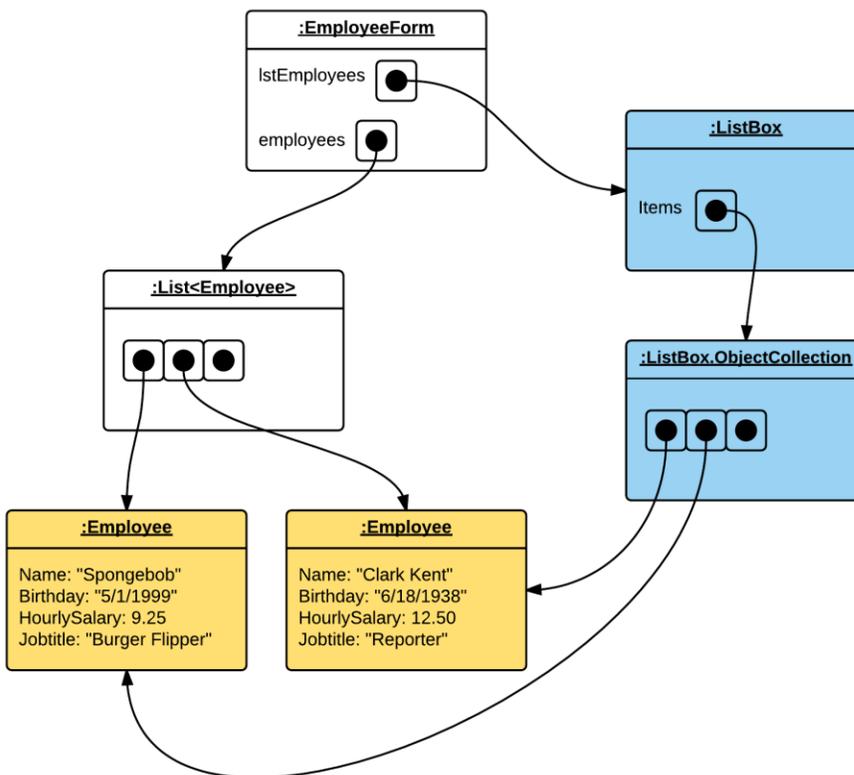
Lines 50–53 update the object in the selected item of the ListBox. It is important to note that if the selected object from the ListBox is updated, there is no need to add it to the List, because both the ListBox and List refer to the same set of objects, as explained below through Figure 13-16. When you update an object from the ListBox, the List that refers to the same object is also updated.

Line 64 clears out any existing items from the ListBox, and line 66 then adds all the Employee objects in the employees List to the ListBox. The AddRange method requires an array, but the List class conveniently provides a ToArray method that returns the contents of the List as an Array. Clearing the ListBox and Adding the entire List to the ListBox refreshes the display of the updated object in the ListBox.

Run the program again and you should be able to manipulate existing objects and add new ones.

You can refer to Figure 13-16 to help you understand which objects are in the system and define the relationships between them. Each box in this diagram represents an object in the system at a point in time (and represents the situation shown in Figure 13-8). The diagram starts with the object that represents the form itself. This object has a reference to the lstEmployees ListBox (among many other references to all the user interface elements) as well as to the employees List. The ListBox has a reference called Items to a ListBox.ObjectCollection, which in turn references the two Employee objects, just like the List does. What this shows is that when the objects are inserted into both the ListBox and the List, you can retrieve from either one and work on the object. Any changes made to the object will be reflected anywhere the object is referenced, since there is only a single copy of the object.

Figure 13-16: Object diagram showing objects in the system



It's time to practice! Do Steps 2-9 and 2-10.

Step 2-9: Add the ability to delete an Employee from the list.

Step 2-10: Change the Employee class to store both a first and last name. Then change the rest of the code to reflect this change. (Add fields to the form and the code that interacts with the form and the Employee objects.)

13.5 Calling Methods (Sending Messages to Objects)

The primary way to interact with an object is to call a method on it or, said in another way, to send a message to that object. Imagine you have an Order object. You could then send messages to that object asking for the customer who placed the order, the total value of the order (calculated by the value of each of the related order line objects), the products in the order, etc. Because the order is an object that knows its own state, you can ask any question related to that state—or the state of related objects.

As an example, we could send a message to an Employee object to add another pay period in this way:

```
emp1.AddPayPeriod(payPeriod);
```

This line of code calls the method `AddPayPeriod` on an Employee object represented by the `emp1` variable passing a parameter represented by the variable `payPeriod`. When calling a method on an object, you always start with a reference to the object followed by a dot and then the method name and a pair of parentheses surrounding any parameter values.

If a method returns an object, you can then immediately call a method on that return value. This is sometimes called *dotting*. Here's an example:

```
emp1.ToString().ToUpper();
```

In this example, the `ToString` method is called on `emp1`. The resulting string is then used for the call to `ToUpper`, which transforms the resulting string to uppercase letters. There is no limit to how many times you can dot your way to making calls. However, it is sometimes advantageous to create intermediate variables, like this:

```
string empString = emp1.ToString();  
string upper = empString.ToUpper();
```

While this code does take up more space, it can be easier to read, and it allows you to examine the value assigned to the intermediate result when debugging.

In the next tutorial you will see how you can call methods on objects and create methods in forms that can be called.

Tutorial 3: Calling Methods and Passing Data between Forms

When testing a program, have you found it tedious to have to enter a new employee every time you launch the program to test some new aspect of it? This can be fixed by hard coding a few Employee objects and passing them as parameters to the form to make them available every time you launch the program. It should be noted that hard coding data generally makes programs difficult to maintain. Here we use it to show how to pass data to a form.

This tutorial also illustrates how the Form class is just like any other class and can be modified to your liking using object-oriented techniques. We will illustrate this in the tutorial by having you modify the constructor for the Form class to accept parameters. This will allow you to select a value on one Form and pass that value to another form to be displayed. You will also see how to pass values back from a child form opened from a parent form. This can be done by creating a public method in the parent form and passing a reference to the parent object to the constructor of the child form.

Passing a List of Objects to a Form

A List of Employee objects can be passed to EmployeeForm by adding the List as a parameter to the constructor of the form.

Step 3-1: Modify Program.cs to create Employee objects, add them to a List, and pass the List to EmployeeForm.

Open the Program.cs class and add the following lines of code at the beginning of the Main method:

```
List<Employee> employees = new List<Employee>();
DateTime birthday = new DateTime(1999, 5, 1);
employees.Add(new Employee("Spongebob", birthday, 10.5m, "Burger flipper"));
birthday = new DateTime(1995, 4, 18);
employees.Add(new Employee("Squidward Tentacles", birthday, 12.56m, "Cashier"));
```

There isn't much new here. The code creates a List to hold Employee objects and then creates two Employee objects and adds them to the List.

Next, modify the last line of the Main method like this:

```
Application.Run(new EmployeeForm(employees));
```

You have added the employees List as a parameter to the constructor of the EmployeeForm class. This will pass a reference to the List to the constructor so it can be used in the EmployeeForm. You will get a red squiggly line because the constructor hasn't yet been modified to accept a List. This will be fixed in the next step.

Step 3-2: Modify the EmployeeForm constructor to work with the Employee List.

Open EmployeeForm.cs to code view. Previously, you added a field named employees (see Figure 13-14), of type List<Employee>. Now you just need to assign the List<Employee> that is passed from Program.cs (when the EmployeeForm object is created) to the employees field in the EmployeeForm class. Figure 13-17 illustrates the changes that need to be made to the constructor.

Figure 13-17: Modified EmployeeForm constructor

```

13 public partial class EmployeeForm : Form
14 {
15     private List<Employee> employees;
16
17     public EmployeeForm(List<Employee> employees)
18     {
19         InitializeComponent();
20         if (employees == null)
21         {
22             employees = new List<Employee>();
23         }
24         this.employees = employees;
25         lstEmployees.Items.AddRange(employees.ToArray());
26     }

```

This is all pretty simple. Notice that in line 17 a parameter is added to the constructor, so it matches up with what you did in Program.cs. Then, in line 20–23, we ensure that the List has been instantiated. Line 24 assigns the List, which is passed from Program.cs to the employees field. Finally, in line 25, the List is added to the ListBox using the AddRange method that you saw previously in Figure 13-15.

You may get a confusing error message that says something about inconsistent accessibility in the constructor. This is likely because the constructor in the form is public, whereas the Employee class doesn't have an access modifier (public/private). If no access modifier is specified, a class is given the access *protected*, which is in between public and private, and thus the class is less accessible than the form's constructor. Fix this, if necessary, by making the Employee class public.

Run the program and check that the two Employees show up in the form.

Creating a Child Form and Passing Data from Parent Form

Next, we will take a look at having multiple objects interact by introducing a PayPeriod class. The idea is that an Employee object would keep a list of PayPeriod objects, as shown in Figure 13-18.

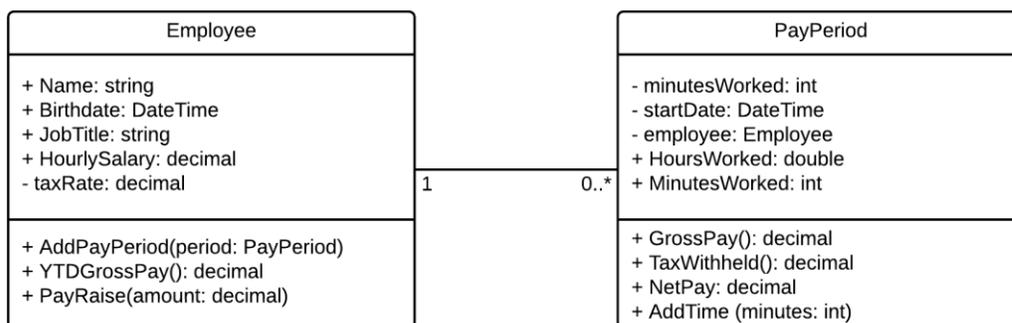
Figure 13-18: Class diagram

Figure 13-19 shows the code to create the PayPeriod class.

Figure 13-19: PayPeriod class

```

 9 public class PayPeriod
10 {
11     //Instance variables:
12     private int minutesWorked;
13     private DateTime startDate;
14     private Employee employee;
15     //Properties:
16     public int MinutesWorked { get { return minutesWorked; } }
17     public decimal HoursWorked { get { return minutesWorked / 60m; } }
18     public DateTime StartDate { get { return startDate; } }
19     public Employee Employee { get { return employee; } }
20     //Constructor:
21     public PayPeriod(int minutesWorked, DateTime startDate, Employee employee)
22     {
23         this.minutesWorked = minutesWorked;
24         this.startDate = startDate;
25         this.employee = employee;
26     }
27     //Methods:
28     public void AddTime(int minutes)
29     {
30         minutesWorked += minutes;
31     }
32
33     public decimal GrossPay()
34     {
35         return HoursWorked * employee.HourlySalary;
36     }
37
38     public decimal TaxWithholding()
39     {
40         return GrossPay() * employee.TaxRate;
41     }
42
43     public decimal NetPay()
44     {
45         return GrossPay() - TaxWithholding();
46     }
47 }

```

This class stores the total number of minutes worked in the pay period in the `minutesWorked` field. This field is modified by the `AddTime` method, which is called for each period worked to add the number of minutes to the pay period. The `HoursWorked` property (line 17) converts `minutesWorked` from integer minutes to decimal hours by dividing the minutes by 60. This way, the implementation is kept as minutes, but any client can request the time in fractional hours instead of minutes.

It's also worth noting that the properties `MinutesWorked` and `HoursWorked` are read-only. The actual time worked can only be changed by calling the `AddTime` method (lines 28–31). The reason for this

restriction is to illustrate a situation where an employee would periodically add more time to their work record. It wouldn't be good if the amount of time could just be set to some arbitrary number. This approach is commonly used in transaction-oriented systems.

The properties `StartDate` and `Employee` (lines 18–19) are also read-only but cannot be changed once they are initialized in the constructor (lines 21–26).

The relationship between the two classes is implemented through the `employee` field (line 14). The data type for this variable is the `Employee` class. This means that each `PayPeriod` object has a variable that points to an `Employee` object. This allows each employee to have multiple pay periods.

Step 3-3: Add `PayPeriod` class.

Add the code shown in Figure 13-19 to create a `PayPeriod` class in the project.

Step 3-4: Modify `Employee` class to hold `PayPeriod` objects.

See Figure 13-20 for what to modify in the `Employee` class in the next steps.

First, add a private field called `payPeriods`, as shown in line 18. Initialize the variable in the constructor (line 27). This variable is a `List` that can hold multiple `PayPeriod` objects. This allows each `Employee` object to easily access all the employee's pay periods.

Figure 13-20: `Employee` class with `List` of `PayPeriods`

```
 9 public class Employee
10 {
11     public string Name { get; set; }
12     public DateTime Birthday { get; set; }
13     public decimal HourlySalary { get; set; }
14     public string JobTitle { get; set; }
15     private decimal taxRate;
16     public decimal TaxRate { get { return taxRate; } }
17
18     private List<PayPeriod> payPeriods;
19
20     public Employee(string name, DateTime birthday, decimal hourlySalary, string jobTitle)
21     {
22         this.Name = name;
23         this.Birthday = birthday;
24         this.HourlySalary = hourlySalary;
25         this.JobTitle = jobTitle;
26         taxRate = 0.25m;
27         payPeriods = new List<PayPeriod>();
28     }
29
30     public void AddPayPeriod(PayPeriod payPeriod)
31     {
32         payPeriods.Add(payPeriod);
33     }
34
35     public override string ToString()
36     {
37         return string.Format("Name: {0}, Birthday: {1:d}, Hourly Salary: {2:c}, Job title: {3}",
38             Name, Birthday, HourlySalary, JobTitle);
39     }
```

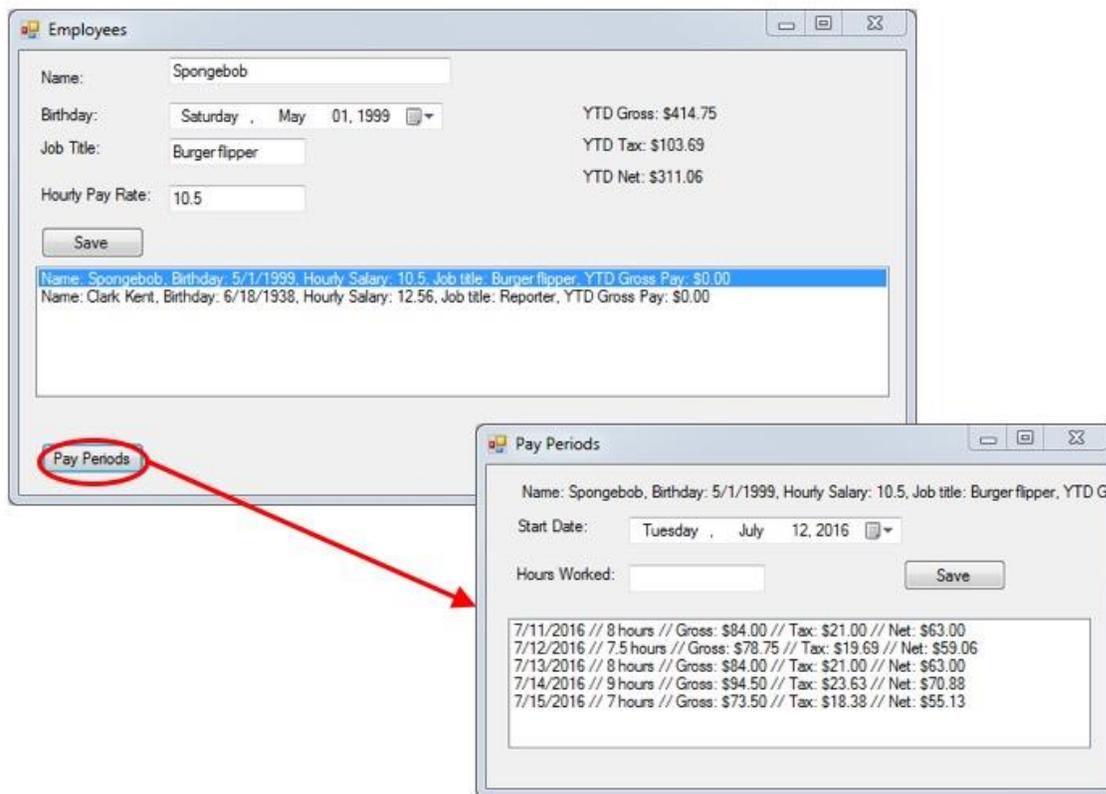
Step 3-5: Add method to add PayPeriods to Employees.

Add the AddPayPeriod method as shown in lines 30–33. This method calls the Add method on the List object to add a new pay period.

Passing Data to Child Form

Next, we will take a look at how you can work with multiple forms as objects and pass data back and forth between them. Figure 13-21 shows a new form named PayPeriodForm that will be launched when the user clicks a button on the Employees form. The information about the selected Employee will be passed to the PayPeriodForm, where the pay periods can be added to the employee.

Figure 13-21: Employee and PayPeriodForm forms

**Step 3-6:** Create PayPeriod form.

Start by adding a new form named PayPeriodForm to the project.

Add the controls you see in Figure 13-21. The controls are named as follows: lblEmployee, dtpStartDate, txtHoursWorked, btnSave, and lstPayPeriods.

Launching a child form

To launch the PayPeriodForm from Employee form, you need to add a button to the Employee form and add the code shown in Figure 13-22 to the Click event handler of the button.

Figure 13-22: Launching the PayPeriodForm from the Employee form

```

76 private void btnPayPeriods_Click(object sender, EventArgs e)
77 {
78     Employee employee = (Employee)lstEmployees.SelectedItem;
79     if (employee != null)
80     {
81         PayPeriodForm payPeriodForm = new PayPeriodForm(employee, this);
82         payPeriodForm.Show();
83     }
84 }

```

The code will create a PayPeriodForm object in line 81 if an employee has been selected. Two parameters are passed to the constructor: references to the selected employee as well as a reference to the EmployeeForm object (this). Those two references will be used in the PayPeriodForm object, as you will see later. You have a syntax error in the call to the PayPeriodForm constructor in line 81, because by default the constructor doesn't take any parameters. You will fix this by modifying the PayPeriodForm constructor. Line 82 makes the Pay Period Form visible.

Step 3-7: Add a button to Employee Form, and add the code to launch the PayPeriodForm.

Switch to the EmployeeForm and add a button named btnPayPeriods with the text Pay Periods. To launch the PayPeriodForm, double click the button in the designer and add the code shown in Figure 13-22 to the generated method.

Step 3-8: In this step, you will add parameters to the constructor of PayPeriodForm, and add code to display the current employee's information in PayPeriodForm and save PayPeriod objects in the current employee object. In addition, you will add code to display all pay periods for the current employee in a ListBox on the PayPeriodForm.

Switch to the code for PayPeriodForm.cs and add private fields and adjust the constructor as shown in Figure 13-23.

Figure 13-23: PayPeriodForm constructor

```

13 public partial class PayPeriodForm : Form
14 {
15     private Employee employee;
16     private EmployeeForm parent;
17
18     public PayPeriodForm(Employee employee, EmployeeForm parent)
19     {
20         InitializeComponent();
21         this.employee = employee;
22         this.parent = parent;
23         lblEmployee.Text = employee.ToString();
24     }
25 }

```

Next, switch to design view and double click the Save button. Then enter the code shown in Figure 13-24 to save a PayPeriod in the current employee object.

Figure 13-24: Saving a new PayPeriod

```

26 private void btnSave_Click(object sender, EventArgs e)
27 {
28     decimal hours;
29     if (Decimal.TryParse(txtHours.Text, out hours))
30     {
31         DateTime startDate = dtpStartDate.Value;
32         int minutesWorked = (int)Math.Round(hours * 60);
33         PayPeriod pp = new PayPeriod(minutesWorked, startDate, employee);
34         employee.AddPayPeriod(pp);
35
36         updateListBox();
37     }
38 }

```

The method reads a fractional number of hours and converts to minutes (line 32) for storing in the PayPeriod object. Line 33 creates a PayPeriod object, and line 34 adds that object to the employee that the form is working on. Notice how you can call a method on an object and pass parameters to the method.

The call to updateListBox in line 36 is throwing an error because the method hasn't been created yet; this will happen in the next step.

Step 3-9: Update the ListBox.

Hover over updateListBox() and click the lightbulb; then select the Generate method option. This will generate a new method. Replace the body of the method with the code shown in Figure 13-25.

Figure 13-25: Updating the ListBox

```

39
40 private void updateListBox() {
41     lstPayPeriods.Items.Clear();
42     foreach (PayPeriod pp in employee.PayPeriods)
43     {
44         string pString = string.Format("{0:d} // {1} hours // Gross: {2:c} // Tax: {3:c} // Net: {4:c}",
45             pp.StartDate, pp.HoursWorked, pp.GrossPay(), pp.TaxWithholding(), pp.NetPay());
46         lstPayPeriods.Items.Add(pString);
47     }
48 }

```

The core of this method is a foreach loop that goes through all the PayPeriod objects in the employee and adds a string for the pay period to the ListBox. There is one problem, as you can see—the PayPeriods property doesn't yet exist in the Employee class, so switch to the Employee class and add the following read-only property:

```
public List<PayPeriod> PayPeriods { get { return payPeriods; } }
```

Run the program and select an employee then add some pay periods to each of the employees.

It's time to practice! Do the following step:

Step 3-10: Modify the constructor in PayPeriodForm to display any existing pay periods for an employee when the form is first loaded.

Sending Messages to the Parent Form

You have seen how to pass data from a parent form to a child form as a parameter in the constructor. But what about the other way—how can you pass information back or call methods in the parent form if needed? Passing data back to the parent form is as simple as modifying a reference to an object on the child form. If the parent form also has a reference to the object, then the changes will be available on the main form without any further action. This is what you did in the previous part of the tutorial; every time you added a `PayPeriod` to an `Employee` on the child `PayPeriodForm`, that change was automatically reflected on the parent `EmployeeForm` because both the parent and child forms had references to the `Employee` object.

The following is an example of how to call a method on a parent form. Some of the work is already done. To launch the `PayPeriodForm` from `Employee` form, you added a button to the `Employee` form, and added the code shown in Figure 13-22 to the `Click` event handler of the button.

In Figure 13-22 and Figure 13-23, you set up a reference in the child form to the parent form. That reference can then be used to access any public method or property from the child form.

What you will do in this tutorial is calculate the average gross income for all employees year-to-date and then display the difference between the average and the income for the employee that is currently displayed on the `PayPeriod` form. To do this, you first need to add a method to the `Employee` class that will calculate the YTD gross income for one employee:

```
public decimal YTDGross()
{
    decimal result = 0m;
    foreach (PayPeriod p in payPeriods)
    {
        if (p.StartDate.Year == DateTime.Now.Year)
        {
            result += p.GrossPay();
        }
    }
    return result;
}
```

This method follows a fairly standard pattern for working with collections of objects. It uses a `foreach` loop to go through all the objects in the collection and then compares the year of the start date to the current year. If they match, the pay period's gross pay is added to a temporary variable, whose value is returned when the loop has gone through all the pay periods.

Step 3-11: Calculate the year to date gross income for an employee.

Add the code to create the `YTDGross` method shown above.

Next, call this method to compute the average gross income.

Step 3-12: Calculate average gross income.

Add the following method to EmployeeForm.cs:

```
public decimal YTDAverageGross()
{
    decimal sum = 0m;
    foreach (Employee e in employees)
    {
        sum += e.YTDGross();
    }
    return sum / (decimal)employees.Count;
}
```

This method follows a similar pattern by going through all the Employee objects in the employee collection and calling the method you just created to calculate the average gross income.

Step 3-13: Display averages on the PayPeriod form.

Add a Label below the ListBox on the PayPeriod form and name it lblCompareAverage. Leave the Text property blank.

Add lines 49–50 in Figure 13-26 to the updateListBox method in PayPeriodForm.cs.

Figure 13-26: Comparing the averages

```
40 private void updateListBox()
41 {
42     lstPayPeriods.Items.Clear();
43     foreach (PayPeriod pp in employee.PayPeriods)
44     {
45         string pString = string.Format("{0:d} // {1} hours // Gross: {2:c} // Tax: {3:c} // Net: {4:c}",
46             pp.StartDate, pp.HoursWorked, pp.GrossPay(), pp.TaxWithholding(), pp.NetPay());
47         lstPayPeriods.Items.Add(pString);
48     }
49     decimal average = parent.YTDAverageGross();
50     lblCompareAverage.Text = string.Format("{0:c}", employee.YTDGross() - average);
51 }
```

Line 49 makes a call on the parent object to the YTDAverageGross() method that you created in EmployeeForm. Line 50 then calls the YTDGross() method that you created in the Employee class and calculates the difference between the two values, which is then displayed on the form.

Run the program and observe how the average is calculated and updated as you switch between the forms, and add pay periods to each of the employees.

It's time to practice! Do exercises 13.8, 13.9, and 13.10 at the end of the chapter.

The following figures contain the full code for the project in Tutorial 3.

Figure 13-27: Program.cs

```

static class Program
{
    /// <summary> The main entry point for the application. </summary>
    [STAThread]
    static void Main()
    {
        List<Employee> employees = new List<Employee>();
        DateTime birthday = new DateTime(1999, 5, 1);
        employees.Add(new Employee("Spongebob", birthday, 10.5m, "Burger flipper"));
        birthday = new DateTime(1995, 4, 18);
        employees.Add(new Employee("Squidward Tentacles", birthday, 12.56m, "Cashier"));

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new EmployeeForm(employees));
    }
}

```

Figure 13-28: Employee.cs

```

public class Employee
{
    public string Name { get; set; }
    public DateTime Birthday { get; set; }
    public decimal HourlySalary { get; set; }
    public string JobTitle { get; set; }
    private decimal taxRate;
    public decimal TaxRate { get { return taxRate; } }

    private List<PayPeriod> payPeriods;
    public List<PayPeriod> PayPeriods { get { return payPeriods; } }

    public Employee(string name, DateTime birthday, decimal hourlySalary, string jobTitle)
    {
        this.Name = name;
        this.Birthday = birthday;
        this.HourlySalary = hourlySalary;
        this.JobTitle = jobTitle;
        taxRate = 0.25m;
        payPeriods = new List<PayPeriod>();
    }

    public void AddPayPeriod(PayPeriod payPeriod)
    {
        payPeriods.Add(payPeriod);
    }

    public override string ToString()
    {
        return string.Format("Name: {0}, Birthday: {1:d}, Hourly Salary: {2:c}, Job Title: {3}",
            Name, Birthday, HourlySalary, JobTitle);
    }

    public decimal YTDGross()
    {
        decimal result = 0m;
        foreach (PayPeriod p in payPeriods)
        {
            if (p.StartDate.Year == DateTime.Now.Year)
            {
                result += p.GrossPay();
            }
        }
        return result;
    }
}

```

```

public decimal YTDTax()
{
    return YTDGross() * taxRate;
}

public decimal YTDNet()
{
    return YTDGross() - YTDTax();
}
}

```

Figure 13-29: EmployeeForm.cs

```

public partial class EmployeeForm : Form
{
    private List<Employee> employees;

    public EmployeeForm(List<Employee> employees)
    {
        InitializeComponent();
        if (employees == null)
        {
            employees = new List<Employee>();
        }
        this.employees = employees;
        lstEmployees.Items.AddRange(employees.ToArray());
    }

    private void btnSave_Click(object sender, EventArgs e)
    {
        //Grab values from the form
        string name = txtName.Text;
        DateTime birthday = dtpBirthday.Value;
        string title = txtJobTitle.Text;
        decimal hourlyRate = Decimal.Parse(txtPayRate.Text);
        Employee employee;
        if (lstEmployees.SelectedIndex >= 0)
        {
            //If an employee is selected in the ListBox, get reference
            //to that employee and update the properties
            employee = (Employee)lstEmployees.SelectedItem;
            employee.Name = name;
            employee.Birthday = birthday;
            employee.JobTitle = title;
            employee.HourlySalary = hourlyRate;
        }
        else
        {
            //If no employee is selected, create a new Employee object
            //based on values in the form.
            employee = new Employee(name, birthday, hourlyRate, title);
            //lstEmployees.Items.Add(employee); //Add new object to ListBox
            employees.Add(employee); //Add new object to collection
        }
        //Clear all items from the ListBox
        lstEmployees.Items.Clear();
        //Add all items from the employees collection to the ListBox
        lstEmployees.Items.AddRange(employees.ToArray());
    }

    private void lstEmployees_SelectedIndexChanged(object sender, EventArgs e)
    {
        Employee employee = (Employee)lstEmployees.SelectedItem;
        txtName.Text = employee.Name;
        txtPayRate.Text = employee.HourlySalary.ToString();
        dtpBirthday.Value = employee.Birthday;
        txtJobTitle.Text = employee.JobTitle;
    }
}

```

```

private void btnPayPeriods_Click(object sender, EventArgs e)
{
    Employee employee = (Employee)lstEmployees.SelectedItem;
    if (employee != null)
    {
        PayPeriodForm payPeriodForm = new PayPeriodForm(employee, this);
        payPeriodForm.Show();
    }
}

public decimal YTDAverageGross()
{
    decimal sum = 0m;
    foreach (Employee e in employees)
    {
        sum += e.YTDGross();
    }
    return sum / (decimal)employees.Count;
}
}

```

Figure 13-30: PayPeriod.cs

```

public class PayPeriod
{
    //Instance variables:
    private int minutesWorked;
    private DateTime startDate;
    private Employee employee;
    //Properties:
    public int MinutesWorked { get { return minutesWorked; } }
    public decimal HoursWorked { get { return minutesWorked / 60m; } }
    public DateTime StartDate { get { return startDate; } }
    public Employee Employee { get { return employee; } }
    //Constructor:
    public PayPeriod(int minutesWorked, DateTime startDate, Employee employee)
    {
        this.minutesWorked = minutesWorked;
        this.startDate = startDate;
        this.employee = employee;
    }
    //Methods:
    public void AddTime(int minutes)
    {
        minutesWorked += minutes;
    }

    public decimal GrossPay()
    {
        return HoursWorked * employee.HourlySalary;
    }

    public decimal TaxWithholding()
    {
        return GrossPay() * employee.TaxRate;
    }

    public decimal NetPay()
    {
        return GrossPay() - TaxWithholding();
    }
}

```

Figure 13-31: PayPeriodForm.cs

```

public partial class PayPeriodForm : Form
{
    private Employee employee;
    private EmployeeForm parent;

    public PayPeriodForm(Employee employee, EmployeeForm parent)
    {
        InitializeComponent();
        this.employee = employee;
        this.parent = parent;
        lblEmployee.Text = employee.ToString();
        updateListBox();
    }

    private void btnSave_Click(object sender, EventArgs e)
    {
        Decimal hours;
        if (Decimal.TryParse(txtHoursWorked.Text, out hours))
        {
            DateTime startDate = dtpStartDate.Value;
            int minutesWorked = (int)Math.Round(hours * 60);
            PayPeriod pp = new PayPeriod(minutesWorked, startDate, employee);
            employee.AddPayPeriod(pp);
            updateListBox();
        }
    }

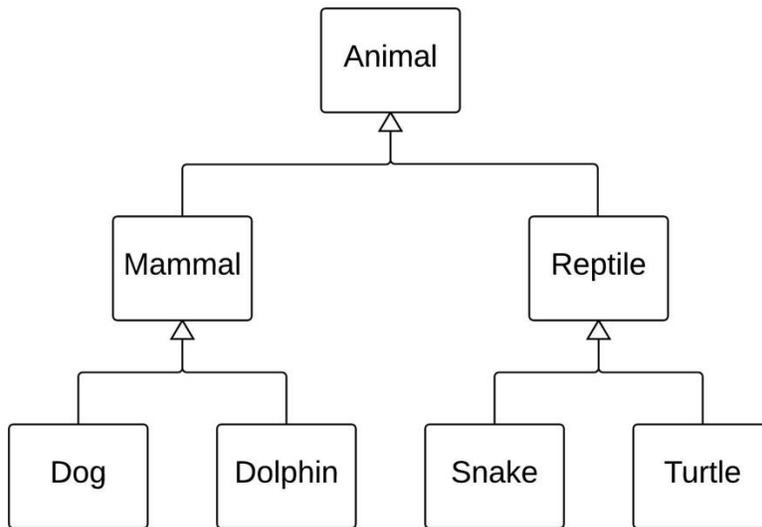
    private void updateListBox()
    {
        lstPayPeriods.Items.Clear();
        foreach (PayPeriod pp in employee.PayPeriods)
        {
            string pString = String.Format("{0:d} // {1} hours // Gross: {2:c} // Tax: {3:c} // Net: {4:c} // ",
                pp.StartDate, pp.HoursWorked, pp.GrossPay(), pp.TaxWithholding(), pp.NetPay());
            lstPayPeriods.Items.Add(pString);
        }
        decimal average = parent.YTDAverageGross();
        lblCompareAverage.Text = string.Format("{0:c}", employee.YTDGross() - average);
    }
}

```

13.6 Introduction to Inheritance

In programming, we often need to modify already created code to add additional features or behaviors. For instance, having created a payroll system with an `Employee` class, we might recognize that there are significant differences in how salary is calculated between salaried and hourly employees. While we could add a variable to the class to hold the type of employee, the calculations would become rather complex, with many if statements checking on which type the employee is. A better approach is to use the concept of inheritance.

Inheritance allows us to extend the functionality of an existing class by inheriting from that existing class. This allows us to modify some parts of a class and keep other parts intact. Conceptually, inheritance allows us to describe relationships between different classes. We can use inheritance to express that some entity is a specialized version of another entity. For instance, we can say that a `Dog` is a specialized version of `Mammal`, which itself is a specialized version of `Animal`. In this way, we create a *hierarchy* of classes (see Figure 13-32).

Figure 13-32: Inheritance hierarchy

As you look at the relationship between two adjacent classes in the hierarchy, you can see the lower one is a more specialized version of the class above it—a Snake is a Reptile with additional characteristics that are not common to all Reptiles. Similarly, a Reptile is an Animal. Of course, it follows from this that Snakes are Animals. This relationship is often referred to as an *is-a* relationship (the lower class *is-a* higher class). We also say that the higher-level class is a *superclass* of the lower-level class, which is a *subclass*. In C#, the superclass is also called a *base class*. In UML diagramming, the inheritance relationship is shown with an empty triangle pointing to the superclass.

The *is-a* relationship exists between classes, not between objects. It expresses that all Dolphins are Mammals and thus have all the properties and behaviors described by Mammal. The alternative to inheritance is *object composition* or aggregation, where one object is made up of other objects. For instance, a car might be said to be made up of four wheels, an engine, two or more doors, etc. Object composition is implemented with a variable in one class, holding a reference to one or more objects of a different class as you saw with the relationship between Employee and PayPeriod in the previous section. In object composition, multiple objects exist to represent the relationship.

However, with inheritance, only a single object is created, even when the object is of one of the subclasses. If a Dog object is created, there will not be a Mammal object or Animal object. However, the Dog will contain all the properties and methods implemented in the Mammal and Animal classes.

Review Questions

- 13.9 Indicate whether the following concepts represent inheritance. If so, which is the superclass and which is the subclass?
- Oven and Kitchen
 - Rectangle and Square
 - Vehicle and Truck
 - Batman and Superhero
 - Student and Person

- Student and Employee
- Order and Customer
- Ferrari and Engine

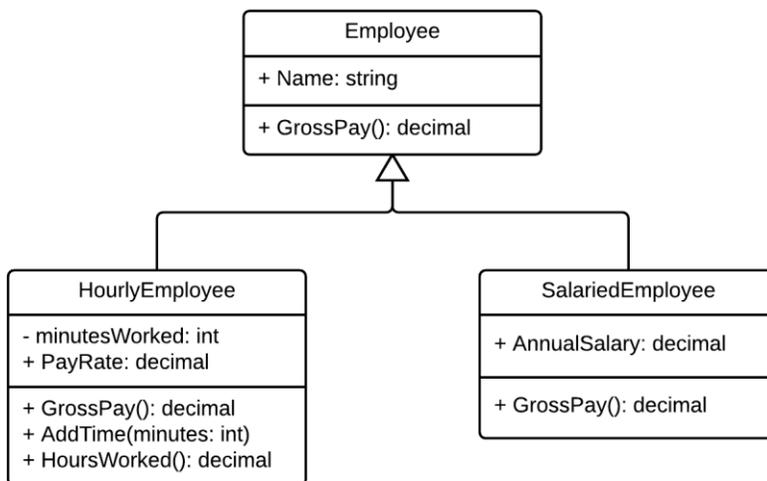
13.10 Identify at least two subclasses for each of the following classes:

- Car
- Student
- Order
- Account
- Book
- Person

13.7 Implementing Inheritance

Returning to our example, consider the Employee class and two subclasses, as shown in Figure 13-33. This simplified hierarchy has two subclasses of Employee: HourlyEmployee and SalariedEmployee. We will use this hierarchy to calculate the gross pay (before any deductions, taxes, etc.) for each of the two types of employees. Hourly Employees' gross pay is calculated as the number of hours worked multiplied by the hourly pay rate, whereas the gross pay for Salaried Employees is calculated based on the annual salary and the number of days in the pay period. To keep things simple, we will use a two-week pay period and assume fifty-two weeks in a year.

Figure 13-33: Employee inheritance hierarchy



When implementing inheritance, you specify that the subclass inherits from the superclass as shown in line 1 of Figure 13-34. The superclass is specified after the class name, separated by a colon.

Any constructor in a subclass must call a superclass constructor. This is done in the signature line of the constructor, where you put a colon and then the method name base (line 11). Notice that parameters show that this constructor in Subclass takes a string and an integer as parameters. The int parameter is then passed to the constructor in Superclass, which would use it to initialize some field declared in SuperClass. We can thus infer from this that Superclass has a constructor that takes a single integer. If you need to call

a specific method in the superclass, you can use the key word **base** in a similar way to the key word **this**, as shown in line 18.

Figure 13-34: Implementing a subclass

```
9      class Subclass: Superclass
10     {
11         public Subclass(string p1, int p2) : base(p2)
12         {
13
14         }
15
16         public void Method()
17         {
18             base.SuperMethod();
19         }
20     }
```

The next tutorial will show how to implement the employee Inheritance hierarchy.

Tutorial 4: Creating Subclasses

In this tutorial you will create a superclass and several of its subclasses to give you some good experience with inheritance.

Step 4-1: Create a project and classes.

Start by creating a new Visual Studio Windows Forms project named Payroll-Inheritance. Once the project is created, add the following three classes by right clicking the project folder in Solution Explorer: Employee, HourlyEmployee, and SalariedEmployee.

Step 4-2: Implement Employee class.

Refer to Figure 13-35 for implementation of the Employee class. This class is very simple, with an automatic property for name and a constructor that initializes the name property.

Figure 13-35: Employee class code

```
9      class Employee
10     {
11         public string Name { get; set; }
12
13         public Employee(string name)
14         {
15             this.Name = name;
16         }
17     }
```

Step 4-3: Set up inheritance in SalariedEmployee.

Switch to SalariedEmployee class and implement the class as shown in Figure 13-36.

You'll note that this class does not have a property to store a name, but as you will see later, this is available through the `Employee` class. Line 9 specifies that `Employee` is the superclass of this class. Note that the `SalariedEmployee` class doesn't have a property for name.

Line 12 has the call to the constructor in `Employee`, passing the name parameter on to the superclass. Notice the difference between the two sets of parentheses in line 12: The first one declares two parameters—that is, it specifies that this constructor takes a string and a decimal. However, the second set contains a value passed to the constructor in `Employee`. Even though it looks a little strange with that “base” key word, this is just a regular call to the superclass constructor. As such, we could have specified any string for the parameter passed to the superclass, or even written one, like “Smith,” directly in the code. It is common that, in addition to taking parameters for its own fields, a subclass constructor also takes all the parameters of its superclass and simply passes them through.

Figure 13-36: `SalariedEmployee` code

```

 9  class SalariedEmployee : Employee
10  {
11      public decimal AnnualSalary { get; set; }
12      public SalariedEmployee(string name, decimal salary) : base(name)
13      {
14          this.AnnualSalary = salary;
15      }
16  }
```

Step 4-4: Instantiate objects.

Switch to the `Program.cs` and add the following two lines of code at the beginning of the `Main` method, which will create two separate objects: an `Employee` object and a `SalariedEmployee` object.

```

Employee emp1 = new Employee("Spongebob");
SalariedEmployee salEmp2 = new SalariedEmployee("Clark Kent", 45000);
```

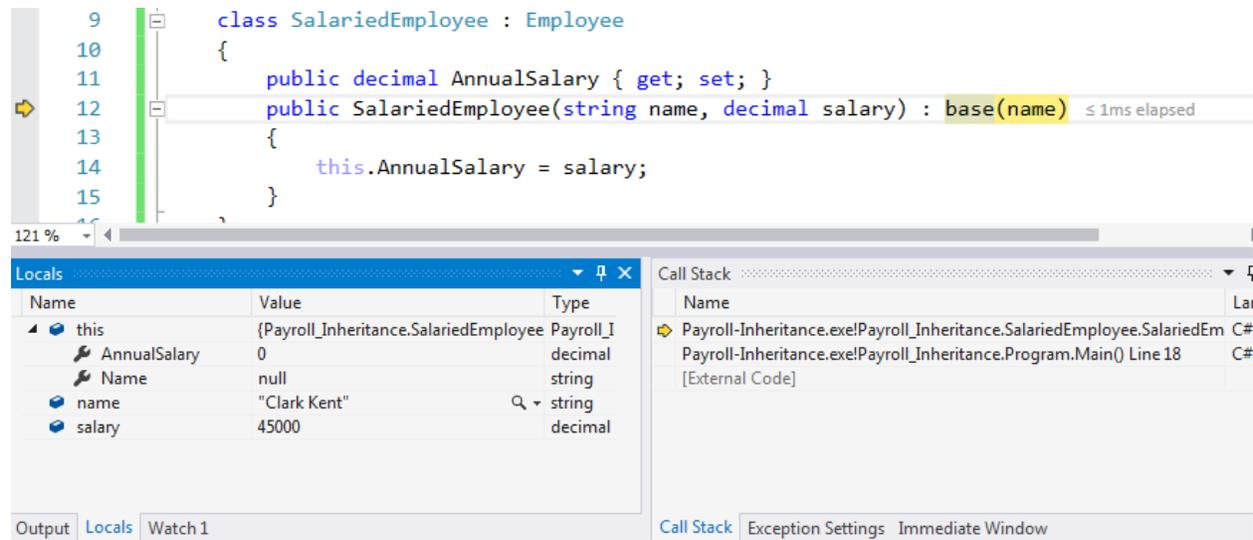
It is assumed that `Spongebob` is neither a salaried nor an hourly employee. If, for example, `Spongebob` was a salaried employee, you won't create a superclass (`Employee`) object for `Spongebob`. You only would create a subclass (`SalariedEmployee`) object, as in the case of `Clark Kent`.

Add a breakpoint in the first of the two lines above and start the debugger (F5). Once you hit the breakpoint, click on `Step Into` (F11) and you should find that execution has moved to the `Employee` class. You can step through the program and notice that everything works as it normally does when creating an object. The string “`Spongebob`” gets assigned to the `Name` property inside the constructor.

Continue stepping until you get back to `Main`; then step into the constructor for `SalariedEmployee`. In the constructor, execution will move to the end of line 12 as shown in `Figure 13-37`. Pay attention to the values in the name variables. They should be the same—namely, “`Clark Kent`” as was entered when the object was created in the `Main` method. If you expand `this` in the `Locals` window, you should also see that `Name` is defined in this object, even if it isn't in the code of `SalariedEmployee`. However, because it is in the superclass, it shows up

here. It's important to note that when a `SalariedEmployee` object is created, there isn't a corresponding `Employee` object. With inheritance, only one object is created, but this object contains all the properties of its superclass.

Figure 13-37: Debugging `SalariedEmployee`



Keep clicking **Step Into** in the debugger until you find yourself in the `Employee` class where the `Name` property is set to `Clark Kent`. Expand `this` again, and note that even though you are in the `Employee` class, you can still see the `AnnualSalary` property, because you are really in the middle of creating a `SalariedEmployee` object, and not an `Employee` object. As you continue, you will return to the `SalariedEmployee` constructor where `AnnualSalary` is initialized to 45,000.

When you are back in the `Main` method, examine the two objects, carefully noting that both the `Employee` object and the `SalariedEmployee` object contain a `Name` property but only `SalariedEmployee` contains an `AnnualSalary` property. Once you have examined the two objects, you can click the **Stop debugging** button.

Repeat the debugging until you are comfortable with the relationship between the classes and objects created.

Now that you have a good understanding of how subclasses work, we add a second subclass to see how two different subclasses inherit common elements from their superclass but can add their own special elements.

Step 4-5: Implement `HourlyEmployee`.

Use the class diagram in Figure 13-33 as a guide in implementing the `HourlyEmployee` class (code is in Figure 13-38).

Start by making the class a subclass of `Employee` (line 9). Then add the `PayRate` property and the private field, `minutesWorked`. These are the pieces of data needed to be stored for the `HourlyEmployee` objects, which are different from both `Employee` and `SalariedEmployee`.

Next, following the example of what you did with `SalariedEmployee`, create the constructor (lines 14–18), passing the name to the superclass constructor and setting the property and field values.

Finally, add the two methods specific to the `HourlyEmployee` class. `AddTime` (lines 20–23) updates the `minutesWorked` field by adding the value of `minutes` parameter to the existing value of `minutesWorked`. `HoursWorked` (lines 25–28) converts the minutes worked and returns as fractional hours worked.

Figure 13-38: Code for `HourlyEmployee` class

```

 9  class HourlyEmployee : Employee
10  {
11      public decimal PayRate { get; set; }
12      private int minutesWorked;
13
14      public HourlyEmployee(string name, decimal payRate) : base(name)
15      {
16          this.PayRate = payRate;
17          this.minutesWorked = 0;
18      }
19
20      public void AddTime(int minutes)
21      {
22          this.minutesWorked += minutes;
23      }
24
25      public decimal HoursWorked()
26      {
27          return minutesWorked / 60.0m;
28      }
29  }

```

Step 4-6: Test `HourlyEmployee` class.

Open `Program.cs` and add the following three lines of code following the creation of the `SalariedEmployee` object that you added previously:

```

HourlyEmployee hourlyEmp3 = new HourlyEmployee("Batman", 40.0m);
hourlyEmp3.AddTime(30);
decimal hours = hourlyEmp3.HoursWorked();

```

The first line creates an `HourlyEmployee` object in the same way that you created a `SalariedEmployee` object previously. The second line calls the `AddTime` method on this object, and the last line calls the `HoursWorked` method on the object.

Remove your previous breakpoint; then place a new breakpoint on the first of these lines, and use the debugger to step into each of the method calls to ensure you understand which code gets called and the result. Note again that no `Employee` object is created and that the `SalariedEmployee` and `HourlyEmployee` objects are entirely independent of each other. Changing the state of the `hourlyEmp3` object has no impact on the state of the `salEmp2` object.

It's also important to note that you cannot call the `AddTime` and `hoursWorked` methods on the `emp1` and `salEmp2` objects, as those methods are declared in the `HourlyEmployee` class.

Try it by adding these lines to the Main method:

```
emp1.AddTime(60);
decimal h2 = emp1.HoursWorked();
salEmp2.AddTime(90);
decimal h3 = salEmp2.HoursWorked();
```

You will get red squiggly lines and error messages saying that Employee and SalariedEmployee don't contain a definition for the methods you're trying to call. This means that you cannot call methods declared in an object's subclasses or an object's sibling classes.

However, you *can* call methods that are declared in a superclass (as long as they are declared public). Comment out the four lines you just added and add these lines instead:

```
string n1 = emp1.Name;
string n2 = salEmp2.Name;
string n3 = hourlyEmp3.Name;
```

You shouldn't get any syntax errors, and if you use the debugger and step through, you will see that n1, n2, and n3 get the Name values for each of the three object (Spongebob, Clark Kent, and Batman).

It's Time to Practice! Do the following steps.

Step 4-7: Create a Person class as a superclass of Employee; then move the Name property to Person.

Note that you should not need to make any changes outside of the Employee and Person classes to have the system function as it did before.

Step 4-8: Add a Customer class as a subclass of Person. Include a property for a Credit Limit. (Each customer may be able to buy up to their credit limit before having to pay.) Create several Customer objects and display on the form.

13.8 Using Subclasses and Superclasses

When you have superclasses and subclasses, you can start doing some more flexible things with variables. For instance, as you have just seen, you can call on properties (and any other public methods) declared in a superclass as if they were declared in the same class as the current object. However, this can be taken further. You can in fact declare a variable to be of a superclass type but then assign that variable an object that is of a subclass type, like this:

```
Employee employee;
employee = new HourlyEmployee("Batman", 40.0m);
employee = new SalariedEmployee("Clark Kent", 45000);
```

Here you see a variable, employee, that is of the declared type Employee, but is assigned first an HourlyEmployee object and then a SalariedObject. When you declare objects like that, you can call any method on the object that is available in the declared type of the variable. So in the above example, you would be able to call on the Name property but not HoursWorked or AddTime—even when it's the HourlyEmployee object that is assigned to the variable. Thus, this method is appropriate when you work with subclass objects and need to use the properties and methods only on the superclass.

One common usage for this is to declare a generic collection to hold objects of a supertype and then at runtime be able to add a mix of subtype objects to the collection and call methods available in the

superclass on all or some of the objects in the collection. For instance, you might have a collection like this, containing the emp1, salEmp2, and se3 objects that were declared in the previous tutorial:

```
List<Employee> employees = new List<Employee>();
employees.Add(emp1);
employees.Add(salEmp2);
employees.Add(hourlyEmp3);
```

With a collection like this, you can have a loop that goes through all the objects in the collection and takes some action for all of them. For instance, if we wanted to create a list of all the names of the employees, we could have a loop like this:

```
string names = "";
foreach(Employee e in employees)
{
    names += e.Name + " ";
}
```

This goes through all the items in the List and then accesses the Name property for each one. Since all the objects in the list are subclasses of Employee, which has the Name property, this will work.

13.9 Overriding Methods

Sometimes a subclass may need to modify the behavior of a method in a superclass. This is called *overriding*. Overriding allows you to specify more specific behavior in a subclass. Often you can specify at a superclass level that a particular behavior is needed, but you cannot specify *how* this behavior will be implemented in a subclass. For instance, in the animal example earlier in the chapter, we could specify that all animals must have a Move behavior, but each kind of animal will have very different ways of moving, so the implementation of the Move behavior will be different in each class. For employees, you will see in the next tutorial that all employees will have a way to calculate how much money they get paid out for a pay period, but the calculation is done differently depending on whether someone is an hourly or salaried employee.

In order to override a method, you write the same method signature in the subclass that you had in the superclass but add the key word *override*. It is common to override the ToString method, which is actually implemented in a class called object—an implicit superclass of every other class. So, in the Employee class, we might have this ToString method:

```
public override string ToString()
{
    return Name;
}
```

The ToString method in HourlyEmployee could look like this:

```
public override string ToString()
{
    return base.ToString() + " Pay Rate: " + PayRate;
}
```

Both of these methods override the version of the method implemented in the superclass. It isn't necessary to have a version in the immediate superclass. If Employee didn't implement ToString, HourlyEmployee could still implement it and would then override the Object version.

Note the call to `base.ToString()` in the `HourlyEmployee` version. This calls the method in the superclass. So in this case, the `HourlyEmployee ToString` method would return the `Name` along with the pay rate. The key word `base` works much like the key word `this`, except that `base` refers to the superclass and `this` refers to the current object. The key word `base` is used in two different ways: as a way to call the superclass constructor from the subclass constructor and as a way to access any method or property in the superclass.

If the `ToString` method is implemented in both the `Employee` class and its subclasses, you might be confused as to which version of the method is actually called at runtime. This is especially confusing if a variable is declared of the supertype but you assign an object of a subtype. For instance, assume you have implemented `ToString` in `Employee` and `HourlyEmployee`, as discussed above, but not in `SalariedEmployee`. What will happen in the following code?

```
Employee employee;
string empStatus;
employee = new HourlyEmployee("Batman", 40.0m);
empStatus = employee.ToString();
employee = new SalariedEmployee("Clark Kent", 45000);
empStatus = employee.ToString();
```

Here we declare a variable `employee` of type `Employee` but then assign an object of type `HourlyEmployee` to that variable (which works because `HourlyEmployee` is a subclass of `Employee`). Then we call `ToString` on the `employee` variable and assign it to the string `empStatus`. After that, we assign a `SalariedEmployee` object and call `ToString` again. While the two lines with the call to `ToString` are identical, they will produce different outputs because the actual objects assigned to `employee` are different.

When you call an overridden method, the runtime environment will always look for the implementation of the method defined in the class in the inheritance hierarchy that is closest to the actual object at runtime. Thus, the first call to `ToString` will call the version in `HourlyEmployee`, but the second version will call the version in `Employee` because `SalariedEmployee` class doesn't override the `ToString` method.

Rules for Overriding Methods

When you create an overridden version of a method in a subclass, you have to follow a few rules:

- The methods must have the same signature—the same method name and the number and types of arguments must be the same.
- The return types have to be the same.
- The access modifier (`public`, `private`, `protected`) has to be the same.
- The overridden method must include the `override` key word.

Overriding versus Overloading

Overriding is often confused with overloading, as both are ways to create different implementations of a method with the same name. The big difference between the two is that overriding only occurs when a subclass implements the same method as a superclass. Overloading, meanwhile, allows you to have methods with the same name in the same class. However, when overloading, you must give the methods different signatures—that is, the number and/or types of parameters for the methods must be different

enough that the compiler can determine which one is intended to be called. With overloading, you can also provide different return types and can have different access modifiers.

Review Questions

- 13.11 If class A is a superclass of class B, which is a superclass of class C, which of the following are allowed?
- `A q = new A();`
 - `A x = new B();`
 - `B y = new A();`
 - `C z = new A();`
- 13.12 Given the classes (A, B, C) in the previous question, explain why the following is allowed:
- ```
List<A> letters = new List<A>();
letters.Add(new A());
letters.Add(new B());
letters.Add(new C());
```
- 13.13 Given the classes (A, B, C) in the previous question, if class B implements a method P that is overridden in C, which version (in B or C) will be called in each of the following code snippets?
- ```
C c = new C();  
c.P();
```
- ```
A a = new B();
a.P();
```
- ```
B b = new C();  
b.P();
```
- 13.14 Given the classes (A, B, C) in the previous question, if class A implements a method Q that is overridden in C (but not in B), which version (in A or C) will be called in each of the following code snippets?
- ```
C c = new C();
c.Q();
```
- ```
A a = new B();  
a.Q();
```
- ```
B b = new C();
b.Q();
```

## Tutorial 5: Implementing the GrossPay Method

Let's return to the Employee example and the GrossPay method, which we discussed earlier. The GrossPay method will calculate the gross pay for a single two-week pay period for an employee. However, this calculation is done differently, depending on whether the employee is an hourly or salaried employee.

When we are done, we would like to be able to add both types of Employees to a collection and calculate the gross pay for all of them, regardless what type they are, with code that might look like this:

```
decimal totalGrossPay = 0m;
foreach (Employee emp in employees)
{
 totalGrossPay += emp.GrossPay();
}
```

We first have to determine which class(es) to implement the GrossPay method in. We have several options:

- In the Employee class only—With this option, we can't distinguish between the different kinds of employees in order to provide a different implementation of the method.
- In the HourlyEmployee and SalariedEmployee classes—This allows us to provide different implementations, but then the loop above wouldn't work and we can't have a collection with both types of employees.

Both of these options could be made to work, but the code becomes much less elegant and harder to maintain. A third option is

- Implement the method in Employee and each of the two subclasses—This allows for collections of both types of employees, and different implementations for the two types of employees, and the loop above will work.

There is only one drawback to this last option: We can't provide a meaningful implementation for the method in the Employee class, since we don't have any information about salary, hours worked, etc. in an object of type Employee. We could simply have this method return a predefined value (perhaps zero or a large negative value) as a signal that the method is undefined, but if the method is used in calculating statistics across many employees, this will likely not be noticed and the statistics will be off.

One solution to this dilemma is to declare that the method is *abstract*. When a method is abstract, no implementation is provided and only the method signature is included in the class. It would look like this for GrossPay in Employee:

```
public abstract decimal GrossPay();
```

Notice the key word `abstract`, indicating that the method is abstract, with the semicolon at the end of the method indicating there is no body to this method.

**Step 5-1:** Add GrossPay method in Employee class.

Go ahead and add the abstract GrossPay method as shown above to the Employee class. You will get a red squiggly line under the GrossPay method name. This is because once you have an abstract method in a class, the class itself must also be declared abstract.

When a class is abstract, you cannot create objects from this class. But in this case, that is exactly what we want. We were unable to provide an implementation of the GrossSalary method in the Employee class because we didn't have any data to allow us to make the calculation. This is a strong indication that we would never create Employee objects, but always either HourlyEmployee or SalariedEmployee objects—that is, we assume that every employee is salaried or hourly.

**Step 5-2:** Make Employee abstract.

Add the abstract key word to the class declaration line, like this:

```
public abstract class Employee
```

**Step 5-3:** Add GrossPay to the SalariedEmployee class.

Switch to SalariedEmployee.cs and you will notice that this class has an error. This is because when you inherit from an abstract class, which this class does, now that you made Employee abstract, you must either declare the subclass abstract or provide an implementation for all abstract methods.

Add the following method anywhere in the SalariedEmployee class:

```
public override decimal GrossPay()
{
 return AnnualSalary / 52 * 2;
 //Gross pay is simply two weeks of annual salary (52 weeks in a year)
}
```

**Step 5-4:** Implement GrossPay in HourlyEmployee.

Add the following code to HourlyEmployee.cs:

```
public override decimal GrossPay()
{
 return HoursWorked() * PayRate;
}
```

You also need to make sure the Person class's access modifier is set as public, since you made the Employee class public. (A superclass can never be less accessible than any of its subclasses.)

If you build the system now, you may get some errors if you have created any Employee objects anywhere. Go through and comment all those lines out so the system builds. You might find some of these in the Main method in Program.cs.

**Step 5-5:** Test the inheritance structure.

To test the system, we will create a number of hourly and salaried employee objects, then calculate the total and average gross pay across all the employees and output this to Labels on a form. Start by adding the code in Figure 13-39 to the Main method in Program.cs to set up the

data to be used for the test. Note that Form1.cs has been renamed to EmployeeInheritanceForm.cs.

**Figure 13-39:** Creating test data for inheritance

```

14 [STAThread]
15 static void Main()
16 {
17 Random r = new Random(); //random number generator - to be used several times below
18 List<Employee> employees = new List<Employee>();
19 string[] employeeNames = { "Superman", "Green Lantern", "Batman", "Spider Man",
20 "Thor", "Hal Jordan", "Wonder Woman", "Captain America"};
21 foreach(string name in employeeNames)
22 {
23 Employee emp = null;
24 if(name.Length > 7) //create hourly employee
25 {
26 decimal payRate = (decimal)r.NextDouble() * 20 + 10; //Random number between 10 and 30
27 HourlyEmployee hrlyEmp = new HourlyEmployee(name, payRate);
28 hrlyEmp.AddTime(r.Next(30, 480)); //Add random time worked to employee
29 emp = hrlyEmp; //Assign to the Employee variable that is added to the List
30 }
31 else //create salaried employee
32 {
33 decimal salary = r.Next(40000, 80000); //Random integer between 40,000 and 80,000
34 emp = new SalariedEmployee(name, salary);
35 }
36 employees.Add(emp); //Add the employee to the list
37 }
38
39 Application.EnableVisualStyles();
40 Application.SetCompatibleTextRenderingDefault(false);
41 Application.Run(new EmployeeInheritanceForm(employees));
42 }

```

In this code, you start by setting up a random number generator (line 17) that will be used to generate random numbers for test purposes.

The names for the employees are put into an array (lines 19–20) and a loop (lines 21–37) generates an employee object for each name. This object is declared in line 23 and added to the list in line 36. We picked an arbitrary limit of seven characters in the name to determine if the employee is an hourly or salaried employee.

For hourly employees (lines 26–29), the pay rate is determined as a random number between 10 and 30. In line 28, a random number of minutes is added to the employee to have worked. Because we need to call `AddTime` on the hourly employee, we can't use the variable `e`, which is of type `Employee`. As discussed earlier, you can assign an object of a subclass to a variable of a superclass type, but in that case you can only use the methods and properties defined in the superclass. So, we declare a variable of the type `HourlyEmployee` (line 27). Line 29 then assigns the `HourlyEmployee` to the `Employee` variable. This works because `Employee` is a superclass of `HourlyEmployee`.

For salaried employees, we just need to determine a salary, which is determined as a random integer between 40,000 and 80,000.

Line 41 has been modified to call the `EmployeeInheritanceForm` constructor with the employees list as an argument. However, because the constructor hasn't been modified yet, you will get an error.

**Step 5-6:** Modify Form constructor.

Switch to EmployeeInheritanceForm.cs and add the code as shown in line 15, 17, and 19 in Figure 13-40. Line 15 sets up a field to hold the list of employees that is passed into the constructor (line 17) and assigned in line 19.

**Figure 13-40:** Code in EmployeeInheritanceForm

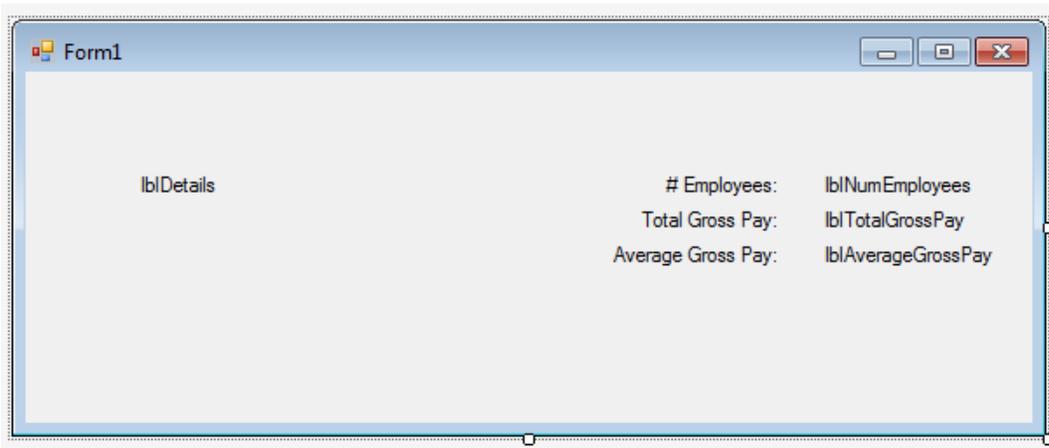
```

13 public partial class EmployeeInheritanceForm : Form
14 {
15 List<Employee> employees;
16
17 public EmployeeInheritanceForm(List<Employee> employees)
18 {
19 this.employees = employees;
20 InitializeComponent();
21
22 lblDetails.Text = employeeDetails();
23 lblNumEmployees.Text = employees.Count.ToString();
24 lblTotalGrossPay.Text = string.Format("{0:c}", totalGrossPay());
25 lblAverageGrossPay.Text = string.Format("{0:c}", averageGrossPay());
26 }
27
28 private decimal totalGrossPay()
29 {
30 decimal result = 0;
31 foreach (Employee e in employees)
32 {
33 result += e.GrossPay();
34 }
35 return result;
36 }
37
38 private decimal averageGrossPay()
39 {
40 return totalGrossPay() / employees.Count;
41 }
42
43 private string employeeDetails()
44 {
45 string result = "";
46 foreach (Employee e in employees)
47 {
48 result += string.Format("Name: {0}, Gross Pay: {1:c}\n", e.Name, e.GrossPay());
49 }
50 return result;
51 }
52 }

```

**Step 5-7:** Set up Form for output.

Open the Form in design view and add seven labels, as shown in Figure 13-41.

**Figure 13-41:** Form design for testing inheritance**Step 5-8:** Implement code for output.

Switch back to the code file and add the three methods shown in lines 28–51 and the rest of the constructor in lines 22–25 in Figure 13-40.

The most interesting thing here is the `totalGrossPay` method, as it illustrates how you can use inheritance to write very simple code that does something fairly powerful. The core of the method is a loop over the list of employees. On each `Employee` object, we call the method `GrossPay`, which returns the gross pay for that employee. However, remember that there is no implementation of the `GrossPay` method in the `Employee` class, so this will call the version in either `SalariedEmployee` or `HourlyEmployee`. Since the employees list contains both salaried and hourly employees, at runtime, the proper version of the method will be chosen based on the actual type of the object in the list. Note that because the `Employee` class is abstract, there could not be any plain `Employee` objects that aren't either salaried or hourly.

**Step 5-9:** Run program to test.

Run the program and you should see output similar to Figure 13-42. Notice that your salary numbers will be different since they are randomly generated.

**It's time to practice! Do the following step:**

**Step 5-10:** To ensure you understand what is happening, set a breakpoint in line 30 of Figure 13-40 and step through the code and into the `GrossPay` method. Notice that you will go into `HourlyEmployee` for some objects and `SalariedEmployee` for other objects.

**Figure 13-42:** Output from testing inheritance

## 13.10 Polymorphism

We have been discussing the concept of *polymorphism*, which means “many forms.” This works in a couple different ways here. First, an object can be considered to be of several different types, both its declared type and the actual type, which could be of a subclass of the declared type, as you saw when you created `HourlyEmployee` and `SalariedEmployee` objects and assigned them to an `Employee` variable.

Second, you also used polymorphism when you defined a method in a superclass and an overridden version in the subclass. You saw how the actual method called was the one in the actual type of the object. For example, in line 33 of Figure 13-40, the `GrossPay` method that called for an hourly employee was the method in the `HourlyEmployee` class and the method that called for a salaried employee was the one in the `SalariedEmployee` class, even though the declared type was `Employee`. The process used to determine the actual method called is called *polymorphic dispatch*.

Polymorphism is one of the most powerful concepts of inheritance. It allows you to create objects of different types and call a method on all of them, but have the actual action taken be different depending on the actual type of the object, as you saw when calculating the total gross pay across all employees above.

We have covered a number of object oriented principles and techniques in this chapter. As you have seen, it allows you to write very powerful and simple code. But we have only scratched the surface of what you can do with it—and have even left a good number of topics out in order to simplify the coverage and keep the page number reasonable in a single chapter. We encourage you to continue studying and using the object oriented material you have learned here to further improve your coding skills.

## Exercises

### Exercise 13.1

Imagine you are creating a computer application to keep track of the work you need to do for your classes. Identify several classes that would be relevant to include in the application, and several objects for each class.

### Exercise 13.2

Choose one of the classes from the previous exercise and identify several attributes and methods for that class.

### Exercise 13.3

Create a new project that can track cars and the repairs on the cars. You will need two classes: Car and Repair. Start by creating a Car class with a few simple properties like Make, Model, and Year. Add a form that can be used to create and modify Car objects and display them in a ListBox.

### Exercise 13.4

Add a Repair class to the project. Repairs have a description and an amount for the repair (e.g., oil change, \$30). Create a form that can be used to add a repair to a selected car.

### Exercise 13.5

Add ability to calculate total amount for all repairs for a car, as well as simple statistics across cars, such as total for all cars, average repair amount, and average per car.

### Exercise 13.6

Add two classes as subclasses of Repair: ListRepair and TimeAndMaterialRepair. A ListRepair is a predefined repair that has a fixed price, so it is similar to Repair in that it simply has a description and an amount. The TimeAndMaterialRepair instead has an amount that is calculated based on the time that a mechanic works on a car and any materials are used in the repair. Add forms to allow for working with both ListRepair and TimeAndMaterialRepairs and associate them with specific cars. By keeping an Amount property in the Repair class, you should not need to modify any of the code calculating statistics related to the amount on each repair. Each of the two classes should also include overridden ToString methods that provide some specifics of the detail of each object.

### Exercise 13.7

Create a Book class that has the following attributes: Author, Title, Year published, and the year of the most recent edition. Add code to ensure that the year published is higher than 1440, when Gutenberg invented the printing press, and no more than one higher than the current year. Also ensure that the year of the most recent edition isn't higher than the published year. Create several Book objects to demonstrate how this works.

**Exercise 13.8**

Add labels to the Employee Form to show the Gross Earnings, Tax Withholdings, and Payment for the currently selected employee.

**Exercise 13.9**

Add more labels to show the difference in Gross Earnings, Tax Withholdings, and Net Payment between the currently selected employee, and the averages for all employees.

**Exercise 13.10**

Add a method to the Employee form that calculates the number of employees with Gross Earnings above the average. Display this value on the form.

**Exercise 13.11**

Rewrite Programming Assignment 2 but use objects to represent an AutoType car and a Rental. You will need to create classes for each of the AutoType and Rental. When you read the data from the database, you pass the data from each record to the constructor of the class in order to create an object. To facilitate saving objects, you can have a method in the class that converts the data of an object to a proper form to be saved and returns it as a string. This string can then be saved in some other part of the program.