

CHAPTER 11



Shopping, Italy

INHERITANCE AND INTERFACES

LEARNING OBJECTIVES

- *STATE THE THREE ESSENTIAL PURPOSES OF INHERITANCE*
- *STATE THE PURPOSE OF A BASE CLASS*
- *STATE THE PURPOSE OF A DERIVED CLASS*
- *STATE THE PURPOSE OF AN ABSTRACT METHOD*
- *STATE THE PURPOSE OF AN ABSTRACT BASE CLASS*
- *DEFINE THE FOLLOWING KEYWORDS: “SEALED”, “VIRTUAL,” “OVERRIDE”, “NEW”, AND “PROTECTED”*
- *DEMONSTRATE YOUR ABILITY TO USE INHERITANCE TO CREATE CLASS HIERARCHIES*
- *DEMONSTRATE YOUR ABILITY TO OVERRIDE BASE CLASS METHODS IN DERIVED CLASSES*
- *STATE THE PURPOSE OF AN INTERFACE*
- *DEMONSTRATE YOUR ABILITY TO CREATE CLASSES THAT IMPLEMENT INTERFACES*
- *STATE THE DEFINITION OF THE TERM POLYMORPHISM*
- *DEMONSTRATE YOUR ABILITY TO WRITE POLYMORPHIC CODE*
- *EXPRESS INHERITANCE RELATIONSHIPS USING UML CLASS DIAGRAMS*

INTRODUCTION

Inheritance is a powerful feature provided by modern object-oriented programming languages. The behavior provided or specified by one class can be adopted or extended by another class. Up to this point you have been using inheritance in every program you have written, although mostly this has been done for you behind the scenes by the C# compiler. For example, every user-defined class you create automatically inherits from the `System.Object` class.

In this chapter, you will learn how to create new derived classes from existing classes and interfaces. There are three ways of doing this: 1) by *extending* the functionality of an existing class, 2) by *implementing* one or more interfaces, or 3) by combining these two methods to create derived classes that both extend the functionality of a base class and implement the operations declared in one or more interfaces.

Along the way I will show you how to create and use abstract methods to create *abstract classes*. You will learn how to create and utilize *interfaces* in your program designs, as well as how to employ the *sealed* keyword to inhibit the inheritance mechanism. You will also learn how to use a Unified Modeling Language (UML) class diagram to show inheritance hierarchies.

By the time you complete this chapter, you will fully understand how to create an object-oriented C# program that exhibits *dynamic polymorphic behavior*. Most importantly, however, you will understand why dynamic polymorphic behavior is a desired object-oriented design objective.

This chapter also builds on the material presented in *Chapter 10 - Compositional Design*. The code examples in this chapter demonstrate the design possibilities you can achieve when you combine inheritance with compositional design.

THREE PURPOSES OF INHERITANCE

Inheritance serves three essential purposes. The first purpose of inheritance is to serve as an object-oriented design mechanism that enables you to think and reason about the structure of your programs in terms of both *generalized* and *specialized* class behavior. A base class implements, or specifies, generalized behavior common to all of its subclasses. Subclasses derived from this base class capitalize on the behavior it provides. Additionally, subclasses may specify, or implement, specialized behavior if required in the context of the design.

When designing with inheritance, you create class hierarchies, where *base classes* that implement *generalized behavior* appear at or near the top of the hierarchy, and *derived classes* that implement *specialized behavior* appear toward the bottom. Figure 11-1 gives a classic example of an inheritance hierarchy showing generalized/specialized behavior.

Referring to figure 11-1 — The `Auto` class sits at the top of the inheritance hierarchy and provides generalized behavior for all of its derived classes. The `Truck` and `Car` classes derive from `Auto`. They provide specialized behavior found only in `Trucks` and `Car` objects. The `DumpTruck` and `PickupTruck` classes derive from `Truck`, which means that `Truck` is also serving as a base class. `DumpTruck` and `PickupTruck` inherit `Truck`'s generalized behavior and also implement the specialized behavior required of these class types. The same holds true for the `PassengerCar` and `SportsCar` classes. Their *direct base class* is `Car`, whose direct base class is `Auto`. For more real world examples of inheritance hierarchies simply consult the .NET API documentation or refer to chapter 5.

The second purpose of inheritance is to provide a way to gain a measure of code reuse within your programs. If you can implement generalized behavior in a base class that's common to all of its subclasses, then you don't have to re-write the code in each subclass. If, in one of your programming projects, you create an inheritance hierarchy and find you are repeating a lot of the same code in each of the subclasses,

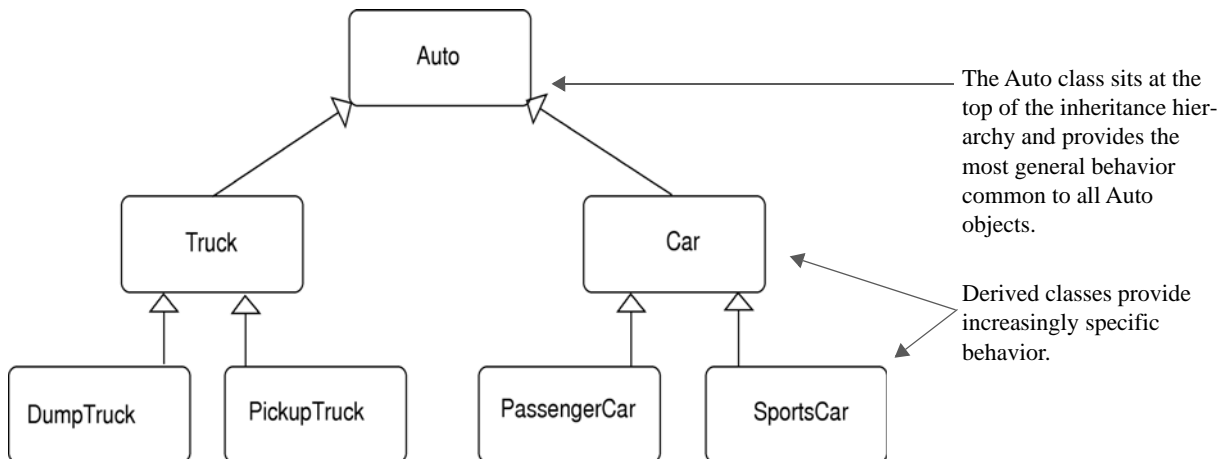


Figure 11-1: Inheritance Hierarchy Illustrating Generalized and Specialized Behavior

then it's time for you to refactor your design and migrate the common code into a base class higher up in your inheritance hierarchy.

The third purpose of inheritance is to enable you to incrementally develop code. It is rare for a programmer, or more often, a team of programmers, to sit down and in one heroic effort completely code the entire inheritance hierarchy for a particular application. It's more likely the case that the inheritance hierarchy, and its accompanying classes, structures, and interfaces, grows over time. The .NET Framework API serves as a prime example.

IMPLEMENTING THE "IS A" RELATIONSHIP

A class that belongs to an inheritance hierarchy participates in what is called an *is a* relationship. Referring again to figure 11-1, a Truck *is an* Auto and a Car *is an* Auto. Likewise, a DumpTruck *is a* Truck, and since a Truck *is an* Auto, a DumpTruck *is an* Auto as well. In other words, class hierarchies are *transitive* in nature when navigating from specialized classes to more generalized classes. They are not transitive in the opposite direction, however. For instance, an Auto is not a Truck or a Car, etc.

A thorough understanding of the *is a* relationships that exist within an inheritance hierarchy will pay huge dividends when you want to substitute a derived class object in code that specifies one of its base classes. This is a critical skill in C#.NET programming and in object-oriented programming in general.

THE RELATIONSHIP BETWEEN THE TERMS TYPE, INTERFACE, AND CLASS

Before moving on, it will help you to understand the relationship between the terms *type*, *interface* and *class*. C# is a strongly typed programming language. This means that when you write a program and wish to call a method on a particular object, the compiler must know, in advance, the type of object to which you refer. In this context, the term *type* refers to *that set of operations or methods a particular object supports*. Every object you use in your programs has an associated type. If, by mistake, you try and call a non-supported method on an object, you will be alerted to your mistake by a compiler error when you try to compile your program.

MEANING OF THE TERM INTERFACE

An *interface* is a construct that introduces a new data type and its set of authorized operations in the form of method, property, event, or indexer declarations. An interface member declaration provides a specification only and no implementation. Interfaces are discussed in more detail later in the chapter.

MEANING OF THE TERM CLASS

A *class* is a construct that introduces and defines a new data type. Like the interface, the class specifies a set of legal operations that can be performed on an object of its type. However, the class can go one step further and provide definitions (*i.e.*, behavior) for some or all of its methods, properties, events, or indexes. A class that provides definitions for all of its members is a *concrete class*, meaning that objects of that class type can be created with the `new` operator. (*i.e.*, They can be *instantiated*.) If a class definition omits the body, and therefore the behavior, of one or more of its members, then that class must be declared to be an *abstract class*. Abstract class objects cannot be created with the `new` operator. I will discuss abstract classes in greater detail later in the chapter.

Quick Review

Inheritance serves three essential purposes: 1) it is an object-oriented design mechanism that enables you to think and reason about your program structure in terms of generalized and specialized class behavior, 2) it provides you with a measure of code reuse within your program by locating common class behavior in base classes, and 3) it provides a means to incrementally develop your programs over time.

Classes that belong to an inheritance hierarchy participate in an *is a* relationship between themselves and their chain of base classes. This *is a* relationship is transitive in the direction of specialized to generalized classes, but not vice versa.

Class and interface constructs are each used to create new, user-defined data types. The interface construct specifies a set of authorized type operations and omits their behavior; the class construct specifies a set of authorized type operations and, optionally, their behavior as well. A class construct, like an interface, can omit the bodies of one or more of its members. Such members must be declared to be abstract. A class that declares one or more abstract members must be declared an abstract class. Abstract class objects cannot be created with the `new` operator.

EXPRESSING GENERALIZATION AND SPECIALIZATION WITH UML

Generalization and specialization relationships can be expressed in a UML class diagram by drawing a solid line with a hollow-tipped arrow from the derived class to the base class, as figure 11-2 illustrates.

Referring to figure 11-2 — `BaseClass` acts as the *direct base class* to `DerivedClass`. Behavior provided by `BaseClass` is inherited by `DerivedClass`. The extent of `BaseClass` behavior that's inherited by `DerivedClass` is controlled by the use of the member access modifiers `public`, `protected`, `internal`, `protected internal`, and `private`. Generally speaking, base class members declared to be `public`, `protected`, `internal`, or `protected internal` are inherited by a derived class. A detailed discussion of how these access modifiers are used to control horizontal and vertical member access is presented later in this chapter. For now, however, let's take a look at an example program that implements the two classes shown in figure 11-2.

A SIMPLE INHERITANCE EXAMPLE

The simple inheritance example program presented in this section expands on the UML diagram shown in figure 11-2. The behavior implemented by `BaseClass` is kept intentionally simple so that you can concentrate on the topic of inheritance. You'll be introduced to more complex programs soon enough.

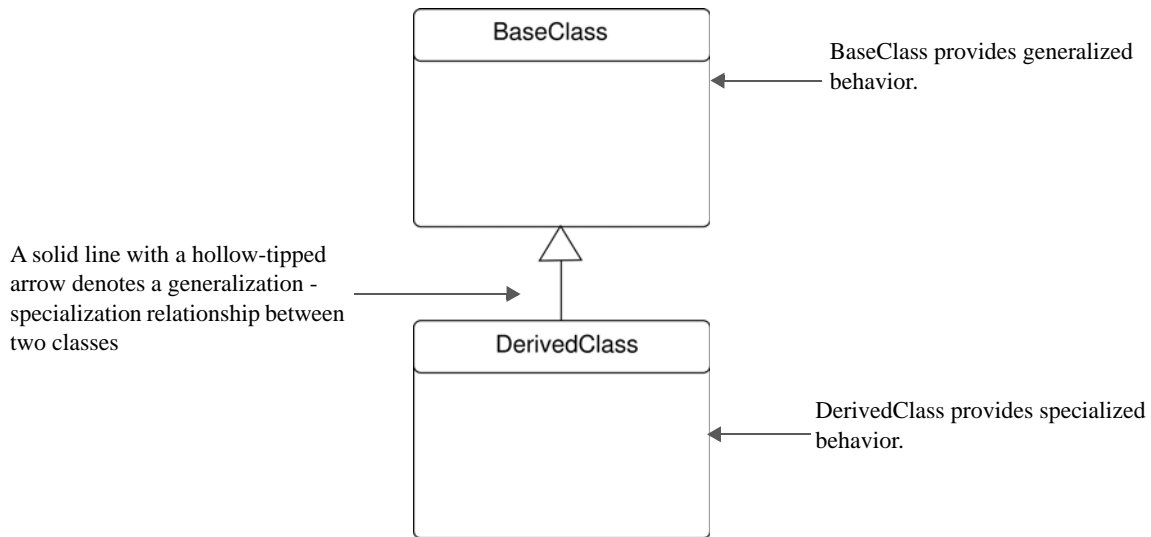


Figure 11-2: UML Class Diagram Showing DerivedClass Inheriting from BaseClass

THE UML DIAGRAM

A more complete UML diagram showing the fields, properties, and methods of BaseClass and DerivedClass is presented in figure 11-3

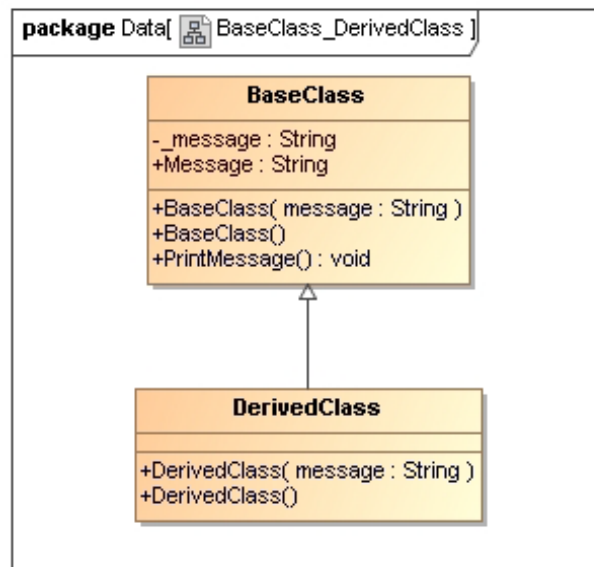


Figure 11-3: UML Diagram of BaseClass and DerivedClass Showing Fields, Properties, and Methods

Referring to figure 11-3 — BaseClass contains one private field named `_message` which is of type `String`. It has one public property named `Message`. BaseClass has three public methods: two constructors and the `PrintMessage()` method. One of the constructors is a default constructor that takes no arguments.

The second constructor has one parameter of type `String` named `message`. Based on this information, objects of type `BaseClass` can be created in two ways. Once an object of type `BaseClass` is created, the `PrintMessage()` method and the `Message` property can be called on that object.

`DerivedClass` has only two constructors that are similar to the constructors found in `BaseClass`. It inherits the public members of `BaseClass`, which include the `Message` property and the `PrintMessage()` method. Let's now take a look at the source code for each class.

BASECLASS SOURCE CODE

11.1 *BaseClass.cs*

```

1  using System;
2
3  public class BaseClass {
4      private String _message;
5
6      public String Message {
7          get { return _message; }
8          set { _message = value; }
9      }
10
11     public BaseClass(String message){
12         Console.WriteLine("BaseClass object created...");
13         Message = message;
14     }
15
16     public BaseClass():this("Default BaseClass message"){ }
17
18     public void PrintMessage(){
19         Console.WriteLine("BaseClass PrintMessage(): " + _message);
20     }
21 }

```

Referring to example 11.1 — `BaseClass` is fairly simple. Its first constructor begins on line 11 and declares one string parameter named `message`. The `_message` field is set via the `Message` property. The default constructor begins on line 16. It calls the first constructor with the string literal “Default BaseClass message!” The `PrintMessage()` method begins on line 18. It simply prints the `Message` property to the console. A `BaseClass` object's message can be changed by setting its `Message` property.

Since the `Message` property and the `PrintMessage()` method each have a body, and are therefore defined, the `BaseClass` is considered a *concrete class*. This means that objects of type `BaseClass` can be created or *instantiated* with the `new` operator.

Example 11.2 gives the code for `DerivedClass`.

DERIVEDCLASS SOURCE CODE

11.2 *DerivedClass.cs*

```

1  using System;
2
3  public class DerivedClass:BaseClass {
4
5      public DerivedClass(String message):base(message){
6          Console.WriteLine("DerivedClass object created...");
7      }
8
9      public DerivedClass():this("Default DerivedClass message"){ }
10 }

```

Referring to example 11.2 — `DerivedClass` inherits the functionality of `BaseClass` by extending `BaseClass`. Note that on line 3, the name `BaseClass` follows the colon character ‘:’. `DerivedClass` itself provides

only two constructors. The first constructor begins on line 5. It declares a string parameter named `message`. The first thing this constructor does is call the string parameter version of the `BaseClass` constructor using the `base()` method with the `message` parameter as an argument. Note how the call to `base()` follows the colon. This is referred to as *constructor chaining*. The next thing the `DerivedClass` constructor does is print a short message to the console.

`DerivedClass`'s default constructor begins on line 9. It calls its version of the string parameter constructor using the string literal *"Default DerivedClass message!"* as an argument to the `this()` call. This ultimately results in a call to the version of the `BaseClass` constructor that takes a string argument.

Let's now take a look at how these two classes can be used in a program.

DRIVERAPPLICATION PROGRAM

11.3 *DriverApplication.cs*

```

1 public class DriverApplication {
2     public static void Main(){
3         BaseClass b1 = new BaseClass();
4         BaseClass b2 = new DerivedClass();
5         DerivedClass d1 = new DerivedClass();
6
7         b1.PrintMessage();
8         b2.PrintMessage();
9         d1.PrintMessage();
10    }
11 }

```

The `DriverApplication` class tests the functionality of `BaseClass` and `DerivedClass`. The important thing to note in this example is which type of object is being declared and created on lines 3 through 5. Starting on line 3, a `BaseClass` reference named `b1` is declared and initialized to point to a `BaseClass` object. On line 4, another `BaseClass` reference named `b2` is declared and initialized to point to a `DerivedClass` object. On line 5, a `DerivedClass` reference named `d1` is declared and initialized to point to a `DerivedClass` object. Note that a reference to a base class object can also point to a derived class object. Also note that this example only uses the default constructors to create each object. This results in the default message text being used upon the creation of each type of object.

Continuing with example 11.3 — On lines 7 through 9, the `PrintMessage()` method is called on each reference. It's time now to compile and run the code. Figure 11-4 gives the results of running example 11.3.

```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\BaseClass_DerivedClass>DriverApplication
BaseClass object created...
BaseClass object created...
DerivedClass object created...
BaseClass object created...
DerivedClass object created...
BaseClass PrintMessage(): Default BaseClass message
BaseClass PrintMessage(): Default DerivedClass message
BaseClass PrintMessage(): Default DerivedClass message
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\BaseClass_DerivedClass>

```

Figure 11-4: Results of Running Example 11.3

As you will notice from studying figure 11-4, there are eight lines of program output that correspond to the creation of the three objects and the three `PrintMessage()` method calls on each reference `b1`, `b2`, and `d1`. Creating a `BaseClass` reference and initializing it to a `BaseClass` object results in the `BaseClass` version (the only version at this point) of the `PrintMessage()` method being called, which prints the default `BaseClass` text message.

Creating a `BaseClass` reference and initializing it to point to a `DerivedClass` object has slightly different behavior. The value of the resulting text printed to the console shows that a `DerivedClass` object was created, which resulted in the `BaseClass _message` field being set to the `DerivedClass` default value. Note

that `DerivedClass` does not have a `PrintMessage()` method, therefore it is the `BaseClass` version of `PrintMessage()` that is called. The `PrintMessage()` method is inherited by `DerivedClass` (*i.e.*, it is accessible to it) because it is declared public.

Finally, declaring a `DerivedClass` reference and initializing it to point to a `DerivedClass` object appears to have the same effect as the previous `BaseClass` reference/`DerivedClass` object combination. This is the case in this simple example because `DerivedClass` simply inherits `BaseClass`'s default behavior and, except for its own constructors, leaves it unchanged.

Quick Review

A base class implements default behavior in the form of public, protected, internal, and protected internal members that can be inherited by derived classes. There are three reference/object combinations: 1) if the base class is a concrete class (meaning it is not abstract) then a base class reference can point to a base class object, 2) a base class reference can point to a derived class object, and 3) a derived class reference can point to a derived class object.

ANOTHER INHERITANCE EXAMPLE: PERSON - STUDENT

Let's now take a look at a more realistic example of inheritance. This example uses the `Person` class presented in chapter 9 as a base class. The derived class will be called `Student`. Let's take a look at the UML diagram for this inheritance hierarchy.

THE PERSON - STUDENT UML CLASS DIAGRAM

Figure 11-5 gives the UML class diagram for the `Student` class inheritance hierarchy. Notice the behavior provided by the `Person` class in the form of its public interface methods and properties. The `Student` class extends the functionality of `Person` and provides a small bit of specialized functionality of its own in the form of the `StudentNumber` and `Major` properties. .

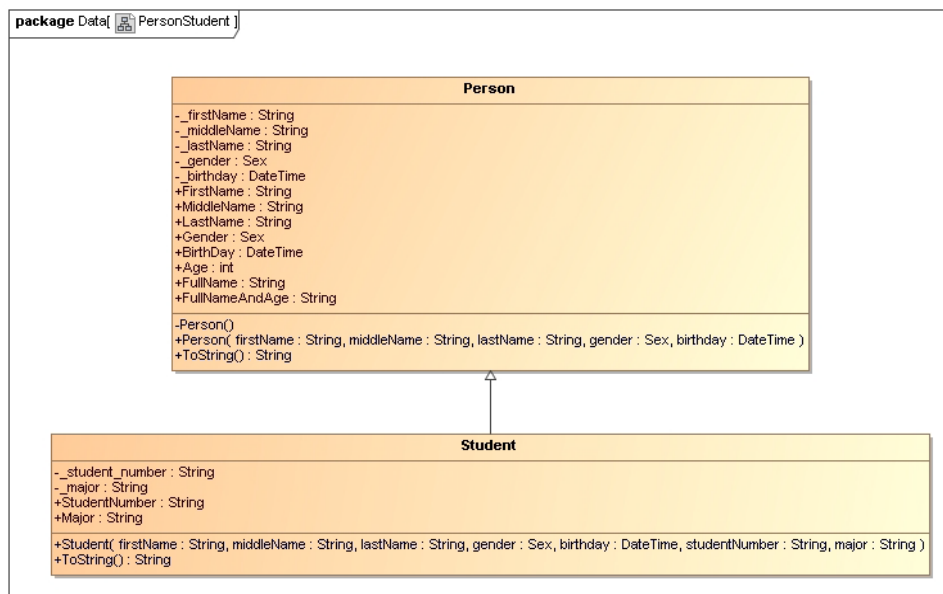


Figure 11-5: UML Diagram Showing Student Class Inheritance Hierarchy

Since the Student class participates in an *is-a* relationship with class Person, a Student object can be used wherever a Person object is called for in your source code. However, now you must be keenly aware of the specialized behavior provided by the Student class, as you will soon see when you examine and run the driver application program for this example.

PERSON - STUDENT SOURCE CODE

11.4 Person.cs

```
1  using System;
2
3  public class Person {
4
5      //enumeration
6      public enum Sex {MALE, FEMALE}
7
8      // private instance fields
9      private String  _firstName;
10     private String  _middleName;
11     private String  _lastName;
12     private Sex      _gender;
13     private DateTime _birthday;
14
15     //default constructor
16     public Person(){
17         _firstName = string.Empty;
18         _middleName = string.Empty;
19         _lastName = string.Empty;
20         _gender = Sex.MALE;
21         _birthday = DateTime.Now;
22     }
23
24     public Person(String firstName, String middleName, String lastName,
25                   Sex gender, DateTime birthday){
26         _firstName = firstName;
27         _middleName = middleName;
28         _lastName = lastName;
29         _gender = gender;
30         _birthday = birthday;
31     }
32
33     // public properties
34     public String FirstName {
35         get { return _firstName; }
36         set { _firstName = value; }
37     }
38
39     public String MiddleName {
40         get { return _middleName; }
41         set { _middleName = value; }
42     }
43
44     public String LastName {
45         get { return _lastName; }
46         set { _lastName = value; }
47     }
48
49     public Sex Gender {
50         get { return _gender; }
51         set { _gender = value; }
52     }
53
54     public DateTime Birthday {
```

```

55     get { return _birthday; }
56     set { _birthday = value; }
57 }
58
59 public int Age {
60     get {
61         int years = DateTime.Now.Year - _birthday.Year;
62         int adjustment = 0;
63         if(DateTime.Now.Month < _birthday.Month){
64             adjustment = 1;
65         }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
66             adjustment = 1;
67         }
68         return years - adjustment;
69     }
70 }
71
72 public String FullName {
73     get { return FirstName + " " + MiddleName + " " + LastName; }
74 }
75
76 public String FullNameAndAge {
77     get { return FullName + " " + Age; }
78 }
79
80 public override String ToString(){
81     return FullName + " is a " + Gender + " who is " + Age + " years old.";
82 }
83
84 } // end Person class

```

The Person class code is unchanged from chapter 9.

11.5 Student.cs

```

1 using System;
2
3 public class Student:Person {
4     private String _studentNumber;
5     private String _major;
6
7     public String StudentNumber {
8         get { return _studentNumber; }
9         set { _studentNumber = value; }
10    }
11
12    public String Major {
13        get { return _major; }
14        set { _major = value; }
15    }
16
17    public Student(String firstName, String middleName, String lastName,
18                  Sex gender, DateTime birthday, String studentNumber,
19                  String major):base(firstName, middleName, lastName, gender, birthday) {
20        _studentNumber = studentNumber;
21        _major = major;
22    }
23
24    public override String ToString(){
25        return (base.ToString() + " Student Number: " + _studentNumber + " Major: " + _major);
26    }
27 } // end Student class definition

```

Referring to example 11.5 — The Student class extends Person and implements specialized behavior in the form of the StudentNumber and Major properties. The Student class has one constructor. With the

exception of the last two parameters, `studentNumber` and `major`, the parameters are those required by the `Person` class. Note how the required person constructor arguments are used in the call to `base()` on line 19. The parameters `studentNumber` and `major` are then used on lines 20 and 21, respectively to set a `Student` object's `_studentNumber` and `_major` fields.

The `Student` class also overrides the `ToString()` method, which begins on line 24. Note how the `Person` class's version of `ToString()` is called via `base.ToString()`. The required additional `Student` information is appended to this string and returned.

This is all the specialized functionality required of the `Student` class for this example. The majority of its functionality is provided by the `Person` class. Let's now take a look at these two classes in action. Example 11.6 gives the test driver program.

11.6 *PersonStudentTestApp.cs*

```

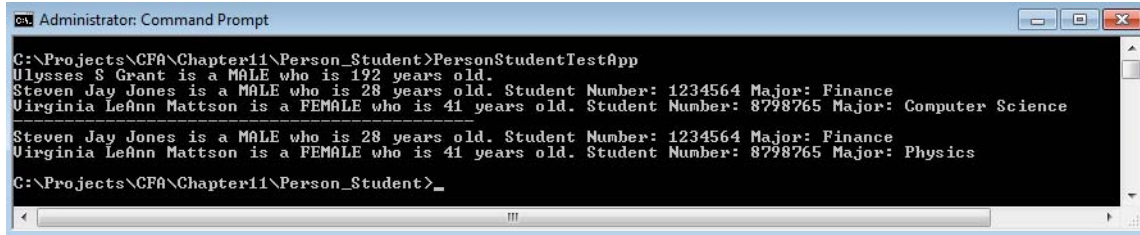
1  using System;
2
3  public class PersonStudentTestApp {
4      public static void Main(){
5          Person p1 = new Person("Ulysses", "S", "Grant", Person.Sex.MALE,
6                                 new DateTime(1822, 04, 22));
7          Person p2 = new Student("Steven", "Jay", "Jones", Person.Sex.MALE,
8                                 new DateTime(1986, 03, 21), "1234564", "Finance");
9          Student s1 = new Student("Virginia", "LeAnn", "Mattson", Person.Sex.FEMALE,
10                                 new DateTime(1973, 09, 14), "8798765", "Computer Science");
11         Console.WriteLine(p1);
12         Console.WriteLine(p2);
13         Console.WriteLine(s1);
14
15         // p2.Major = "Criminal Justice"; // ERROR: p2 is a Person reference
16         s1.Major = "Physics";
17
18         Console.WriteLine("-----");
19         Console.WriteLine(p2);
20         Console.WriteLine(s1);
21     }
22 }

```

Referring to example 11.6 — This program is similar in structure to example 11-3 in that it declares three references and shows you the effects of accessing methods and properties via those references. A `Person` reference named `p1` is declared on line 5 and initialized to point to a `Person` object. On line 7, another `Person` reference named `p2` is declared and initialized to point to a `Student` object. On line 9, a `Student` reference is declared and initialized to point to a `Student` object. On lines 11, 12, and 13, each object's information is written to the console.

Line 15 is commented out. This line, if you were to try to compile it, will cause a compiler error because an attempt is made to set the `Major` property on a `Student` object via a `Person` reference. Now, repeat the previous sentence to yourself several times until you fully understand its meaning. Good! Now, you may ask, and rightly so at this point, "But wait, why can't you set the `Major` property on a `Student` object?" You can, but `p2` is a `Person` type reference, which means that the compiler is enforcing the interface defined by the `Person` class. Remember the "C# is a strongly-typed language..." spiel I delivered earlier in this chapter? I will show you how to use casting to resolve this issue after I show you how this program runs.

Continuing with example 11.6, on line 16, the `Major` property is set on a `Student` object via a `Student` reference. Finally, on lines 19 and 20, the information for references `p2` and `s1` is written to the console. Figure 11-6 gives the results of running example 11.6.



```

cs. Administrator: Command Prompt
C:\Projects\CF\Chapter11\Person_Student>PersonStudentTestApp
Ulysses S Grant is a MALE who is 192 years old.
Steven Jay Jones is a MALE who is 28 years old. Student Number: 1234564 Major: Finance
Virginia LeAnn Mattson is a FEMALE who is 41 years old. Student Number: 8798765 Major: Computer Science
-----
Steven Jay Jones is a MALE who is 28 years old. Student Number: 1234564 Major: Finance
Virginia LeAnn Mattson is a FEMALE who is 41 years old. Student Number: 8798765 Major: Physics
C:\Projects\CF\Chapter11\Person_Student>_

```

Figure 11-6: Results of Running Example 11.6

CASTING

OK, now let's take a look at a modified version of example 11.6 that takes care of the problem encountered on line 15. Example 11.7 gives the modified version of `PersonStudentTestApp.cs`.

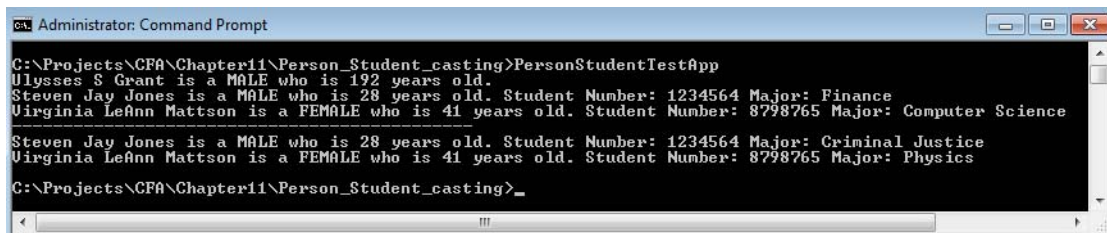
11.7 PersonStudentTestApp.cs (Mod 1)

```

1 using System;
2
3 public class PersonStudentTestApp {
4     public static void Main(){
5         Person p1 = new Person("Ulysses", "S", "Grant", Person.Sex.MALE,
6                                 new DateTime(1822, 04, 22));
7         Person p2 = new Student("Steven", "Jay", "Jones", Person.Sex.MALE,
8                                 new DateTime(1986, 03, 21), "1234564", "Finance");
9         Student s1 = new Student("Virginia", "LeAnn", "Mattson", Person.Sex.FEMALE,
10                                 new DateTime(1973, 09, 14), "8798765", "Computer Science");
11     Console.WriteLine(p1);
12     Console.WriteLine(p2);
13     Console.WriteLine(s1);
14
15     ((Student)p2).Major = "Criminal Justice"; // OK - Person reference is cast to type Student
16     s1.Major = "Physics";
17
18     Console.WriteLine("-----");
19     Console.WriteLine(p2);
20     Console.WriteLine(s1);
21 }
22 }

```

Referring to example 11.7 — Notice on line 15 that the compiler has been instructed to treat the `p2` reference as though it were a `Student` type reference. This form of explicit *type coercion* is called *casting*. Casting only works if the object the reference actually points to is of the proper type. In other words, you can cast `p2` to a `Student` type because it points to a `Student` object. However, you could not cast the `p1` reference to `Student` since it actually points to a `Person` object. Figure 11-7 shows the results of running example 11.7.



```

cs. Administrator: Command Prompt
C:\Projects\CF\Chapter11\Person_Student_casting>PersonStudentTestApp
Ulysses S Grant is a MALE who is 192 years old.
Steven Jay Jones is a MALE who is 28 years old. Student Number: 1234564 Major: Finance
Virginia LeAnn Mattson is a FEMALE who is 41 years old. Student Number: 8798765 Major: Computer Science
-----
Steven Jay Jones is a MALE who is 28 years old. Student Number: 1234564 Major: Criminal Justice
Virginia LeAnn Mattson is a FEMALE who is 41 years old. Student Number: 8798765 Major: Physics
C:\Projects\CF\Chapter11\Person_Student_casting>_

```

Figure 11-7: Results of Running Example 11.7

USE CASTING SPARINGLY

Casting is a helpful feature, but too much casting usually means your design is not optimal from an object-oriented point of view. You will see more situations in this book where casting is required, but mostly, I try to show you how to design programs that minimize the need to cast.

Quick Review

The Person class provided the default behavior for the Student class. The Student class inherited Person's default behavior and implemented its own specialized behavior.

Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without casting as long as the reference type supports the method you are trying to call. Casting forces, or coerces, the compiler into treating a reference to an object of one type as if it were a reference to an object of another. This is extremely helpful in some circumstances but, as a rule, use casting sparingly. Also, casting only works if the object really is of the type you are casting it to.

OVERRIDING BASE CLASS METHODS

So far you have only seen examples of inheritance in which the derived class fully accepted the behavior provided by its base class. This section shows you how to override base class behavior in the derived class by overriding base class methods.

To override a base class method in a derived class you need to redefine the method with the exact signature in the derived class. The overriding derived class method must also return the same type as the overridden base class method and be declared with the keyword `override`. You also need to add the keyword `virtual` to the base class method declaration. By using the `virtual/override` keyword pair, you can achieve polymorphic behavior.

Let's take a look at a simple example. Figure 11-8 gives a UML class diagram for the revised BaseClass and DerivedClass classes.

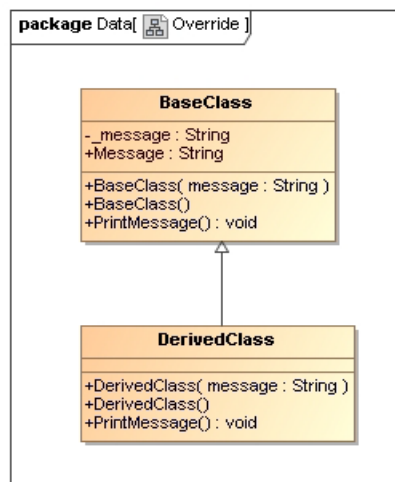


Figure 11-8: UML Class Diagram For BaseClass & DerivedClass

Referring to figure 11-8 — Notice that `DerivedClass` now has a public method named `PrintMessage()`. `BaseClass` has been modified by adding the keyword `virtual` to the declaration of its `PrintMessage()`

method, which is not shown in the diagram. Example 11.8 gives the source code for the modified version of `BaseClass`.

11.8 *BaseClass.cs (Mod 1)*

```

1  using System;
2
3  public class BaseClass {
4      private String _message;
5
6      public String Message {
7          get { return _message; }
8          set { _message = value; }
9      }
10
11     public BaseClass(String message){
12         Console.WriteLine("BaseClass object created...");
13         _message = message;
14     }
15
16     public BaseClass():this("Default BaseClass message"){ }
17
18     public virtual void PrintMessage(){
19         Console.WriteLine("BaseClass PrintMessage(): " + _message);
20     }
21 }

```

Referring to example 11.8 — The only change to `BaseClass` is the addition of the `virtual` keyword to the definition of the `PrintMessage()` method. The `virtual` keyword enables the `PrintMessage()` method to be overridden in `DerivedClass`. If you omit the `virtual` keyword from a base class method or other over-rideable member, then you will get a compiler error if you attempt to override that member in a derived class. Example 11.9 gives the code for the modified version of `DerivedClass`.

11.9 *DerivedClass.cs (Mod 1)*

```

1  using System;
2
3  public class DerivedClass:BaseClass {
4
5      public DerivedClass(String message):base(message){
6          Console.WriteLine("DerivedClass object created...");
7      }
8
9      public DerivedClass():this("Default DerivedClass message"){ }
10
11     public override void PrintMessage(){
12         Console.WriteLine("DerivedClass PrintMessage(): " + Message);
13     }
14 }

```

Referring to example 11.9 — The `DerivedClass`'s version of `PrintMessage()` on line 11 overrides the `BaseClass` version. Note the use of the `override` keyword in the `PrintMessage()` method definition. How does this affect the behavior of these two classes? A good way to explore this issue is to recompile and run the `DriverApplication` given in example 11.3. Figure 11-9 shows the results of running the program using the modified versions of `BaseClass` and `DerivedClass`.

Referring to figure 11-9 — Compare these results with those of figure 11-4. The first message is the same, which is as it should be. The `b1` reference points to a `BaseClass` object. The second message is different, though. Why is this so? The `b2` reference is pointing to a `DerivedClass` object. When the `PrintMessage()` method is called on the `DerivedClass` object via the `BaseClass` reference, the overriding `PrintMessage()` method provided in `DerivedClass` is called. This is an example of *polymorphic behavior*. A base class reference, `b2`, points to a derived class object. You call a method provided by the base class

```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\BaseClass_DerivedClass_Override>DriverApplication
BaseClass object created...
BaseClass object created...
DerivedClass object created...
BaseClass object created...
DerivedClass object created...
BaseClass PrintMessage(): Default BaseClass message
DerivedClass PrintMessage(): Default DerivedClass message
DerivedClass PrintMessage(): Default DerivedClass message
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\BaseClass_DerivedClass_Override>_

```

Figure 11-9: Results of Running Example 11.3 with Modified Versions of BaseClass and DerivedClass

interface via the base class reference, but it's overridden in the derived class and, voila, you have polymorphic behavior. Pretty cool, huh?

Quick Review

Derived classes can *override* base class behavior by providing *overriding methods*. An *overriding method* is a method in a derived class that has the same signature as the base class method it is intending to override. Use the `virtual` keyword to declare an overrideable base class method. Use the `override` keyword to define an overriding derived class method. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

ABSTRACT METHODS AND ABSTRACT BASE CLASSES

An *abstract method* is one that appears in the body of a class declaration but omits the method body. A class that declares one or more abstract methods must be declared to be an abstract class. If you create an abstract method and forget to declare the class as being abstract, the compiler will inform you of your mistake.

Now, you could simply declare a class to be abstract even though it provides implementations for all of its methods. This would prevent you from creating objects of the abstract class directly with the `new` operator. This may or may not be the intention of your application design goals.

THE PRIMARY PURPOSE OF AN ABSTRACT BASE CLASS

The primary purpose of an abstract base class is to provide a set of one or more public interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy. The key phrase is “*expected to be found in some derived class further down the inheritance hierarchy.*” This means that as a designer, you would employ an abstract class in your application architecture when you want a base class to specify rather than implement behavior, and you expect derived classes to actually implement the behavior specified by the base class interface.

OK, why would you want to do this? Why create a class that does nothing but specify a set of interface methods? Good question! The short answer is that abstract classes will constitute the upper tier of your inheritance hierarchy. The upper tier of an inheritance hierarchy is where you expect to find specifications for the general behavior inherited by derived classes, which appear in the lower tier of an inheritance hierarchy. The derived classes, at some point, must provide implementations for those abstract methods specified in their base classes. Designing application architectures in this fashion — abstractions at the top and concrete implementations at the bottom — enables the architecture to be *extended*, rather than *modified*, to accommodate new functionality. This design technique, also referred to as the *Open-Closed Principle*,

injects a good dose of stability into your application architecture. This and other advanced object-oriented design techniques are discussed in more detail in *Chapter 23 — Three Design Principles*.

EXPRESSING ABSTRACT BASE CLASSES IN UML

Figure 11-10 shows a UML diagram that contains an abstract base class named `AbstractClass`.

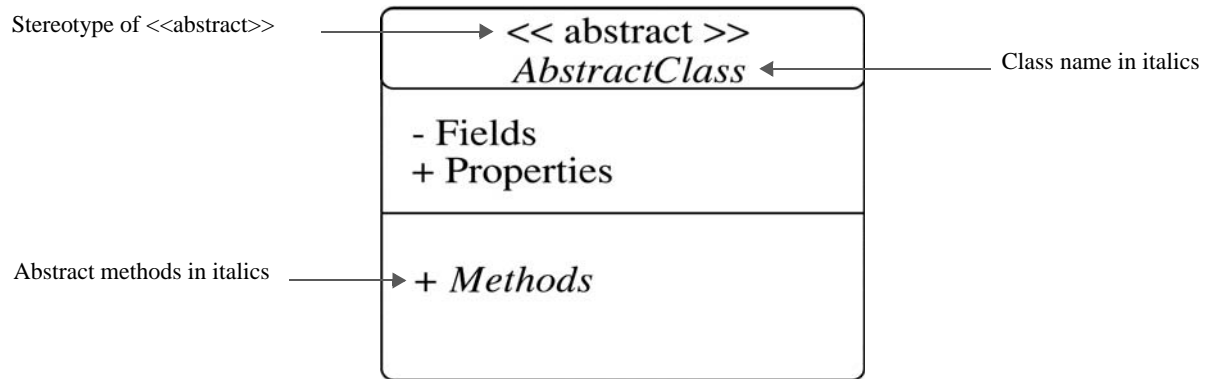


Figure 11-10: Expressing an Abstract Class in the UML

Referring to figure 11-10 — The stereotype `<<abstract>>` is optional, but if you draw your UML diagrams by hand, it’s hard to write in italics, so, this notation comes in handy. Abstract classes can have the same kinds of members as normal classes, but abstract members are shown in italics. Let’s now have a look at a short abstract class inheritance example.

ABSTRACT CLASS EXAMPLE

Figure 11-11 gives the UML class diagram for our example:

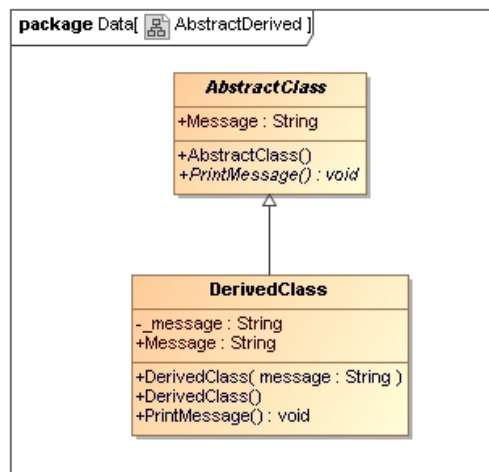


Figure 11-11: UML Class Diagram Showing the `AbstractClass` and `DerivedClass` Inheritance Hierarchy

Referring to figure 11-11 — `AbstractClass` has two methods and one property. The first method is its default constructor and it is not abstract. (Remember, constructors cannot be abstract.) The next method, `PrintMessage()`, is shown in italics and is therefore an abstract method. The `Message` property is also abstract in this example but a limitation in MagicDraw prevents it from being displayed in italics, otherwise, MagicDraw is a fine UML design tool.

DerivedClass inherits from AbstractClass. Since AbstractClass's PrintMessage() method is abstract and has no implementation, DerivedClass must provide an implementation for it. DerivedClass's PrintMessage() method is in plain font, indicating it has an implementation.

Now, if for some reason, you as a designer decided to create a class that inherited from DerivedClass, and you defer the implementation of the PrintMessage() method to that class, then DerivedClass would itself have to be declared to be an abstract class. I just wanted to mention this because in most situations you will have more than the two-tiered inheritance hierarchy I have used here in this simple example.

Let's now take a look at the code for these two classes. Example 11.10 gives the code for AbstractClass.

11.10 AbstractClass.cs

```

1  using System;
2
3  public abstract class AbstractClass {
4
5      public abstract String Message {
6          get;
7          set;
8      }
9
10     public AbstractClass(){
11         Console.WriteLine("AbstractClass object created...");
12     }
13
14     public abstract void PrintMessage();
15 }

```

Referring to example 11.10 — The important point to note is that on line 3 the keyword `abstract` indicates that this is an abstract class definition. The `abstract` keyword is also used on lines 5 and 14 in the `Message` property definition and the `PrintMessage()` method declaration. An abstract member is implicitly virtual, so you don't need to add the `virtual` keyword to an abstract member definition. In fact, doing so will produce a compiler warning. Also note how the `Message` property's `get` and `set` accessors are terminated with a semicolon, indicating they have no implementation. (*i.e.*, No curly braces, no body, no implementation.) The same holds true for the `PrintMessage()` method. Example 11.11 gives the code for `DerivedClass`.

11.11 DerivedClass.cs

```

1  using System;
2
3  public class DerivedClass:AbstractClass {
4      private String _message;
5
6      public override String Message {
7          get { return _message; }
8          set { _message = value; }
9      }
10
11     public DerivedClass(String message){
12         _message = message;
13         Console.WriteLine("DerivedClass object created...");
14     }
15
16     public DerivedClass():this("Default DerivedClass message"){ }
17
18     public override void PrintMessage(){
19         Console.WriteLine("DerivedClass PrintMessage(): " + _message);
20     }
21 }

```

Referring to example 11.11 — `DerivedClass` extends `AbstractClass`. `DerivedClass` provides an implementation for each of `AbstractClass`'s abstract members. Let's take a look now at the test driver program that exercises these two classes.

11.12 *DriverApplication.cs*

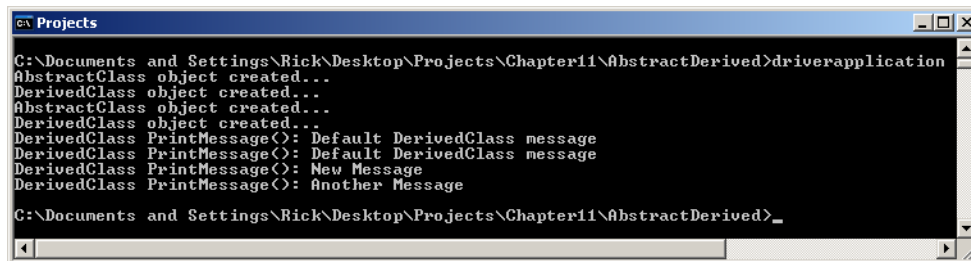
```

1 public class DriverApplication {
2     public static void Main(){
3         AbstractClass a1 = new DerivedClass(); // preferred combination
4         DerivedClass d1 = new DerivedClass();
5         a1.PrintMessage();
6         d1.PrintMessage();
7         a1.Message = "New Message";
8         d1.Message = "Another Message";
9         a1.PrintMessage();
10        d1.PrintMessage();
11    }
12 }
```

Referring to example 11.12 — Remember, you cannot directly instantiate an abstract class object. On line 3, a reference to an `AbstractClass` type object named `a1` is declared and initialized to point to a `DerivedClass` object. On line 4, a reference to a `DerivedClass` type object named `d1` is declared and initialized to point to a `DerivedClass` object.

The comment on line 3 that says “preferred combination” means that the abstract type reference pointing to the derived class object is the preferred combination in an object-oriented program. Remember, your goal is to write code that works the same regardless of what type of object a reference points to. The abstract class serves as a specification for behavior. As long as it points to an object that implements the specified behavior, things should work fine.

Figure 11-12 shows the results of running example 11.12.



```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\AbstractDerived>driverapplication
AbstractClass object created...
DerivedClass object created...
AbstractClass object created...
DerivedClass object created...
DerivedClass PrintMessage(): Default DerivedClass message
DerivedClass PrintMessage(): Default DerivedClass message
DerivedClass PrintMessage(): New Message
DerivedClass PrintMessage(): Another Message
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\AbstractDerived>
```

Figure 11-12: Results of Running Example 11.12

Quick Review

An *abstract member* is a member that omits its body and has no implementation behavior. A class that declares one or more abstract members must be declared to be `abstract`.

The primary purpose of an *abstract class* is to provide a specification for behavior whose implementation is expected to be found in some derived class further down the inheritance hierarchy.

Designers employ abstract classes to provide a measure of application architectural stability.

INTERFACES

An *interface* is a construct that functions like an implicit abstract class. In C#, a derived class can extend the behavior of only one class, but it can implement as many interfaces as it requires. Interfaces themselves can inherit from (*i.e.*, extend) multiple interfaces.

THE PURPOSE OF INTERFACES

The purpose of an interface is to provide a *specification for behavior* in the form of abstract properties, methods, events, and indexers. An interface declaration introduces a new data type, just as class declarations and definitions do.

AUTHORIZED INTERFACE MEMBERS

C# interfaces can contain the following members:

- Non-Static Property, Method, Event, and Indexer declarations: — Implicitly public and abstract and, **beginning with C# 8**, can contain default implementations.
- Static Methods: — Implicitly public and must have an implementation.

THE DIFFERENCES BETWEEN AN INTERFACE AND AN ABSTRACT CLASS

Table 11-1 summarizes the differences between abstract classes and interfaces.

Abstract Class	Interface
Must be declared abstract with the <code>abstract</code> keyword.	Is implicitly abstract.
Can contain abstract and concrete members.	Can contain static and non-static members. All non-static members in an interface are implicitly public and abstract. Beginning with C# 8.0 non-static members can contain default implementations. Static members are implicitly public and must have an implementation.
Can contain fields, constants, and static members.	Can contain declarations and default implementations for properties, methods, events, and indexers. Cannot contain instance data such as fields, auto-properties, or property-like events.
Can contain nested class and interface declarations.	Can contain declarations and default implementations for properties, methods, events, and indexers. Cannot contain instance data such as fields, auto-properties, or property-like events.
Can extend one class and implement many interfaces.	Can extend many interfaces.

Table 11-1: Differences Between Abstract Classes and Interfaces

EXPRESSING INTERFACES IN UML

Interfaces are expressed in UML in two ways as is shown in figure 11-13.

Referring to figure 11-13 — One way to show an interface in UML is by using a circle with the name of the interface close by. The second way involves the use of an ordinary class diagram that includes the stereotype `<<interface>>`. Each of these diagrams, the circle and the class diagram, can represent the use of interfaces in an inheritance hierarchy, as is discussed in the following section.

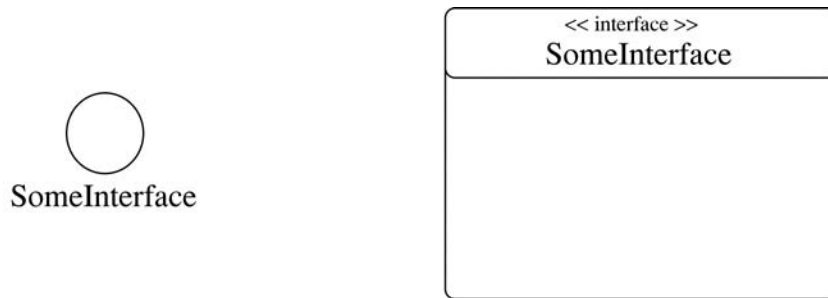


Figure 11-13: Two Types of UML Interface Diagrams

EXPRESSING REALIZATION IN A UML CLASS DIAGRAM

When a class implements an interface it is said to be *realizing* that interface. Interface *realization* is expressed in UML in two distinct forms: 1) the *simple form* in which the circle represents the interface and is combined with an association line to create a *lollipop diagram*, or 2) the *expanded form* in which an ordinary class diagram represents the interface. Figure 11-14 illustrates the use of the lollipop diagram to convey the simple form of realization. Figure 11-15 shows an example of the expanded form of realization.

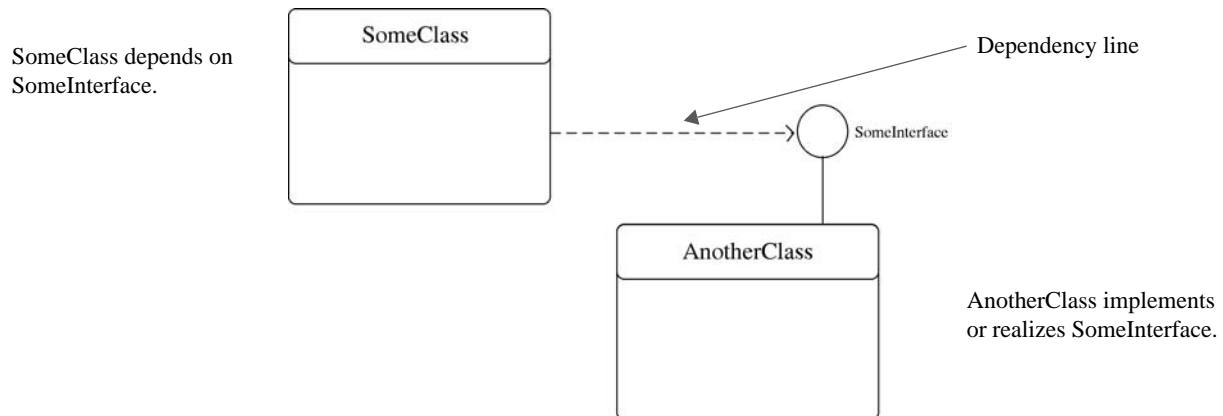


Figure 11-14: UML Diagram Showing the Simple Form of Realization

AN INTERFACE EXAMPLE

Let's turn our attention to a simple example of an interface in action. Figure 11-16 gives the UML diagram of the `IMessagePrinter` interface and a class named `MessagePrinter` that implements the `IMessagePrinter` interface. The source code for these two classes is given in examples 11.13 and 11.14.

11.13 IMessagePrinter.cs

```

1  using System;
2
3  public interface IMessagePrinter {
4      String Message {
5          get;
6          set;
7      }
8
9      void PrintMessage();

```

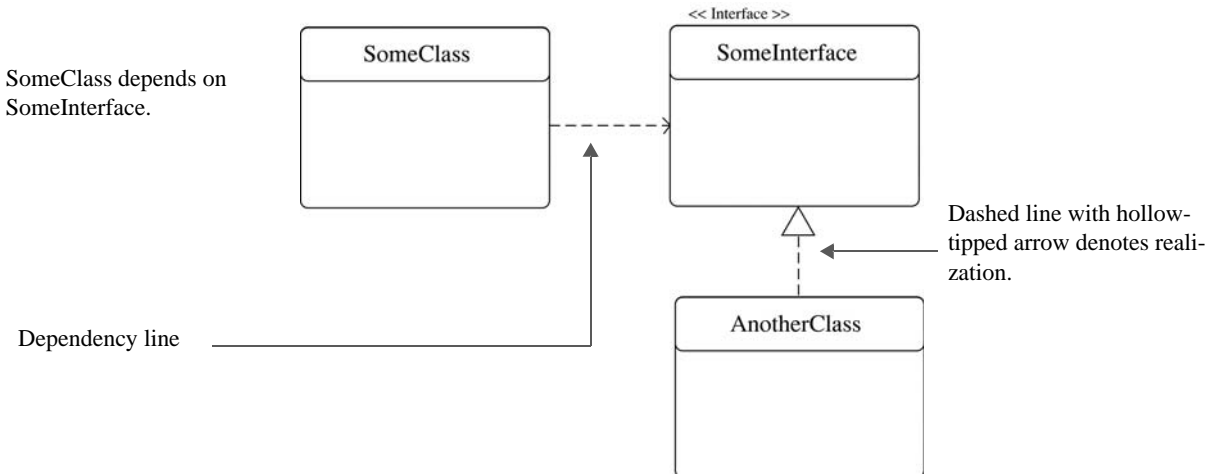


Figure 11-15: UML Diagram Showing the Expanded Form of Realization

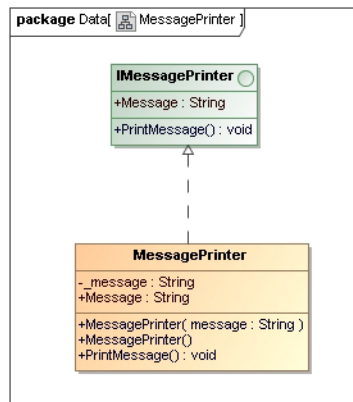


Figure 11-16: UML Diagram Showing the MessagePrinter Class Implementing the IMessagePrinter Interface

```
10 }
```

```

1  using System;
2
3  public class MessagePrinter:IMessagePrinter {
4      private String _message;
5
6      public String Message {
7          get { return _message; }
8          set { _message = value; }
9      }
10
11     public MessagePrinter(String message){
12         _message = message;
13         Console.WriteLine("MessagePrinter object created...");
14     }
15
16     public MessagePrinter():this("Default MessagePrinter message"){ }
17
18     public void PrintMessage(){
19         Console.WriteLine("MessagePrinter PrintMessage(): " + _message);
20     }
21 }
```

11.14 MessagePrinter.cs

11.15 DriverApplication.cs

```

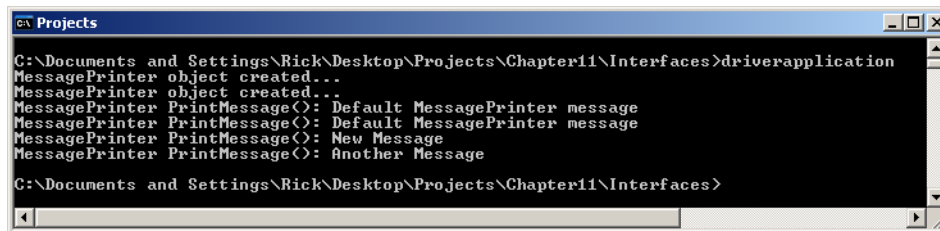
1 public class DriverApplication {
2     public static void Main(){
3         IMessagePrinter i1 = new MessagePrinter();
4         MessagePrinter m1 = new MessagePrinter();
5         i1.PrintMessage();
6         m1.PrintMessage();
7         i1.Message = "New Message";
8         m1.Message = "Another Message";
9         i1.PrintMessage();
10        m1.PrintMessage();
11    }
12 }

```

As you can see from example 11.13, the `IMessagePrinter` interface is short and simple. All it does is declare two interface members: the `Message` property and the `PrintMessage()` method. The implementation of these interface members is left to any class that implements the `IMessagePrinter` interface, as the `MessagePrinter` class does in example 11.14.

Example 11.15 gives the test driver program for this example. As you can see on line 3, you can declare an interface type reference. I called this one `i1`. Although you cannot instantiate an interface directly with the `new` operator, you can initialize an interface-type reference to point to an object of any concrete class that implements the interface. The only object members you can access via the interface-type reference are those members specified by the interface. You could of course cast to a different type if required, as long as the object implements that type's interface, but strive to minimize the need to cast in this manner.

Figure 11-17 gives the results of running example 11.15.



```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Interfaces>driverapplication
MessagePrinter object created..
MessagePrinter object created..
MessagePrinter PrintMessage(): Default MessagePrinter message
MessagePrinter PrintMessage(): Default MessagePrinter message
MessagePrinter PrintMessage(): New Message
MessagePrinter PrintMessage(): Another Message
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Interfaces>

```

Figure 11-17: Results of Running Example 11.15

Quick Review

The purpose of an *interface* is to specify behavior. Interfaces can have four types of members: 1) properties, 2) methods, 3) events, and 4) indexers. Classes can extend only one other class, but they can implement as many interfaces as required. Interfaces can extend as many interfaces as necessary.

CONTROLLING HORIZONTAL AND VERTICAL ACCESS

The term *horizontal access* describes the level of access an object of one type has to the members of another type. I discussed this topic in detail in chapter 9. The term *vertical access* refers to the level of access a derived class has to its base class members. In both cases access is controlled by the access modifiers `public`, `protected`, `private`, `internal`, and `protected internal`.

The default class member access is `private`. That is, when you omit an explicit access modifier from the definition of a class member, the member's accessibility is set to `private` by default. Conversely, interface members are implicitly `public` and no other access specifier can be applied to an interface member

declaration. A derived class does not have access to a base class's private members. (*i.e.*, Private members are not inherited.)

Derived classes have access to their base class's public, protected, internal, and protected internal members. (*i.e.*, These members are inherited by the derived class.)

The most frequently used access modifiers are `private`, `public`, and `protected`. As a rule of thumb you will declare a class's fields and one of more of its methods to be `private`. You saw an example of this already with private fields. Utility methods meant to be used only by their containing class are usually declared to be `private` as well.

If you want a member to be inherited by derived classes (*i.e.*, accessible vertically) but not accessible horizontally by other classes or code, declare it to be `protected`. If you want a member to be both horizontally and vertically accessible to all code within an assembly but only vertically accessible to derived classes outside the assembly, declare it to be `protected internal`.

Quick Review

Use the access modifiers `private`, `protected`, `public`, `internal`, and `protected internal` to control horizontal and vertical member access.

SEALED CLASSES AND METHODS

Sometimes you want to prevent classes from being extended or individual methods of a particular class from being overridden. Use the keyword `sealed` for these purposes. When used to declare a class, it prevents that class from being extended. When used to declare a method in a base class, it prevents the method from being overridden in a derived class.

If the base class method is `virtual`, and is overridden in a derived class, use the `sealed` keyword in conjunction with the `override` keyword to prevent further overriding in another derived class. In other words, a sealed method is an overridden method that you want to prevent from being further overridden in the future.

You cannot use the keyword `sealed` in combination with the keyword `abstract` for obvious reasons.

Quick Review

Use the `sealed` keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

POLYMORPHIC BEHAVIOR

A good definition of *polymorphism* is “*The ability to operate on and manipulate different derived objects in a uniform way.*” (Sadr) Add to this the following amplification: “*Without polymorphism, the developer ends up writing code consisting of large case or switch statements. This is in fact the litmus test for polymorphism. The existence of a switch statement that selects an action based upon the type of an object is often a warning sign that the developer has failed to apply polymorphic behavior effectively.*” (Booch)

Polymorphic behavior is easy to understand. In a nutshell, it is simply the act of using the set of public interface members defined for a particular class (or interface) to interact with that class's (or interface's)

derived classes. When you write code, you need some level of a priori knowledge about the type of objects your code will manipulate. In essence, you have to set the bar at some level, meaning that at some point in your code, you need to make an assumption about the type of objects with which you are dealing and the behavior they manifest. An object's type, as you know, is important because it specifies the set of operations that are valid for objects of that type and its subtypes.

Code that's written to take advantage of polymorphic behavior is generally cleaner, easier to read, easier to maintain, and easier to extend. If you find yourself casting a lot, you are not writing polymorphic code. If you use the *typeof* operator frequently to determine object types, then you are not writing polymorphic code. *Polymorphic behavior is the essence of object-oriented programming.*

Quick Review

Polymorphic behavior is achieved in a program by targeting a set of operations specified by a base class or interface and manipulating their derived class objects via those operations. This uniform treatment of derived class objects results in cleaner code that's easier to extend and maintain. Polymorphic behavior is the essence of object-oriented programming.

INHERITANCE EXAMPLE: EMPLOYEE

This section offers an inheritance example that extends, once again, the functionality of the Person class given in chapter 9. Figure 11-18 gives the UML diagram.

Referring to figure 11-18 — The Employee class extends Person and implements the IPayable interface. However, in this example, as you will see later, the implementation of the Pay() method specified in the IPayable interface is deferred by the Employee class to its derived classes. It does this by mapping the IPayable.Pay() method to an abstract method, which means the Employee class now becomes an abstract class.

The HourlyEmployee and SalariedEmployee classes extend the functionality of Employee. Each of these classes will implement the Pay() method in its own special way.

From a polymorphic point of view, you could write a program that uses these classes in several ways — it just depends on which set of interface methods you want to target. For instance, you could write a program that contains an array of Person references. Each of these Person references could then be initialized to point to an HourlyEmployee object or a SalariedEmployee object. In either case, the only members you can access on these objects via a Person reference without casting are those public members specified by the Person class.

Another approach would be to declare an array of IPayable references. Then again, you could initialize each IPayable reference to point to either an HourlyEmployee object or a SalariedEmployee object. Now the only members of each object that you can access without casting is the Pay() method.

A third approach would be to declare an array of Employee references and initialize each reference to point to either an HourlyEmployee object or a SalariedEmployee object. In this scenario, you could then access any member specified by Person, IPayable, and Employee. This is the approach taken in the EmployeeTestApp program listed in example 11.20.

The code for each of these classes (except Person class, which is unchanged from chapter 9) along with the EmployeeTestApp class is given in examples 11.16 through 11.20.

11.16 IPayable.cs

```
1 public interface IPayable {
2     double Pay();
3 }
```

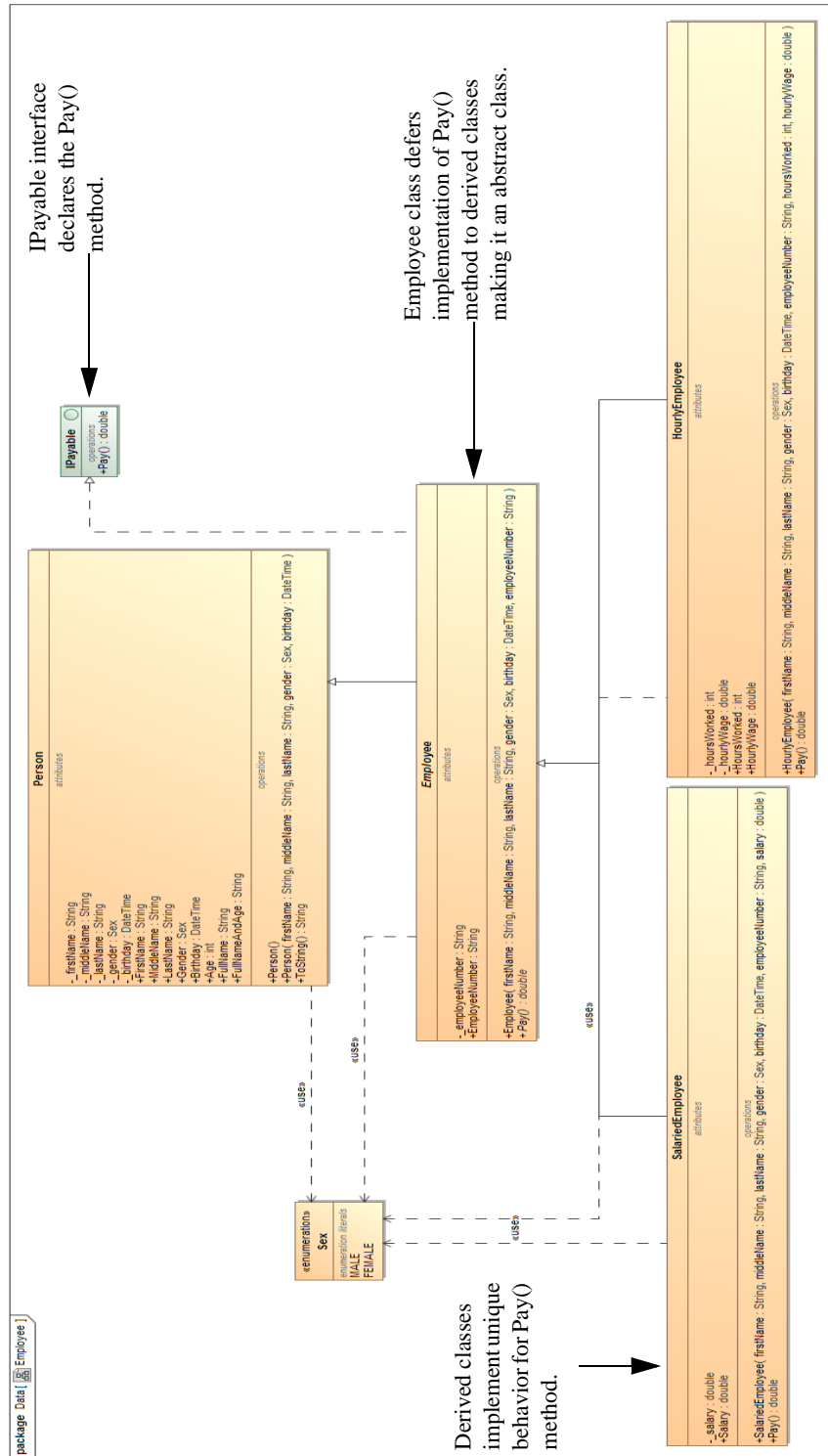



Figure 11-18: Employee Class Inheritance Hierarchy

```

1 using System;
2
3 /*****
4 * The Employee class extends Person and implements
5 * IPayable, but since it defers the actual

```

11.17 Employee.cs

```

6 * implementation of IPayable's Pay() method
7 * to derived classes it must be declared an
8 * abstract class.
9 *****/
10
11 public abstract class Employee : Person, IPayable {
12     private String _employeeNumber;
13
14     public String EmployeeNumber {
15         get { return _employeeNumber; }
16         set { _employeeNumber = value; }
17     }
18
19     public Employee(String firstName, String middleName, String lastName,
20                     Sex gender, DateTime birthday, String employeeNumber):
21         base(firstName, middleName, lastName, gender, birthday){
22         _employeeNumber = employeeNumber;
23     }
24
25     public abstract double Pay(); // map IPayable.Pay() to an abstract method
26                                     // and defer implementation
27 } // end Employee class definition

```

11.18 HourlyEmployee.cs

```

1 using System;
2
3 public class HourlyEmployee : Employee {
4
5     private int _hoursWorked;
6     private double _hourlyWage;
7
8     public int HoursWorked {
9         get { return _hoursWorked; }
10        set { _hoursWorked = value; }
11    }
12
13    public double HourlyWage {
14        get { return _hourlyWage; }
15        set { _hourlyWage = value; }
16    }
17
18    public HourlyEmployee(String firstName, String middleName, String lastName,
19                          Sex gender, DateTime birthday, String employeeNumber, int hoursWorked,
20                          double hourlyWage): base(firstName, middleName, lastName, gender, birthday,
21                          employeeNumber){
22        _hoursWorked = hoursWorked;
23        _hourlyWage = hourlyWage;
24    }
25
26    public override double Pay(){
27        return _hoursWorked * _hourlyWage;
28    }
29 }

```

11.19 SalariedEmployee.cs

```

1 using System;
2
3 public class SalariedEmployee : Employee {
4
5     private double _salary;
6
7     public double Salary {
8         get { return _salary; }
9         set { _salary = value; }

```

```

10 }
11
12 public SalariedEmployee(String firstName, String middleName, String lastName,
13     Sex gender, DateTime birthday, String employeeNumber, double salary):
14     base(firstName, middleName, lastName, gender, birthday, employeeNumber){
15     _salary = salary;
16 }
17
18 public override double Pay(){
19     return _salary/24;;
20 }
21 }

```

11.20 EmployeeTestApp.cs

```

1 using System;
2
3 public class EmployeeTestApp {
4     public static void Main(){
5         Employee[] employees = new Employee[4];
6
7         employees[0] = new HourlyEmployee("Rick", "W", "Miller", Person.Sex.MALE,
8             new DateTime(1964,02,02), "11111111", 80, 17.00);
9
10        employees[1] = new SalariedEmployee("Steve", "J", "Jones", Person.Sex.MALE,
11            new DateTime(1975,08,09), "22222222", 130000.00);
12
13        employees[2] = new HourlyEmployee("Bob", "E", "Evans", Person.Sex.MALE,
14            new DateTime(1956,12,23), "33333333", 80, 25.00);
15
16        employees[3] = new SalariedEmployee("Coralie", "S", "Miller", Person.Sex.FEMALE,
17            new DateTime(1967,11,21), "44444444", 67000.00);
18
19        for (int i=0; i<employees.Length; i++){
20            Console.WriteLine(employees[i].FullName + " " +
21                String.Format("{0:C}", employees[i].Pay()));
22        }
23    } // end Main()
24 } // end class definition

```

Referring to example 11.20 — On line 5, the EmployeeTestApp program declares an array of Employee references named employees. On lines 7 through 17, it initializes each Employee reference to point to either an HourlyEmployee or SalariedEmployee object.

In the for statement on line 19, each Employee object is manipulated polymorphically via the interface specified by the Employee class, which includes the interfaces inherited from Person and IPayable. The results of running this program are shown in figure 11-19.

```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Employee>employeetestapp
Rick W Miller $1,360.00
Steve J Jones $5,416.67
Bob E Evans $2,000.00
Coralie S Miller $2,791.67
C:\Documents and Settings\Rick\Desktop\Projects\Chapter11\Employee>

```

Figure 11-19: Results of Running Example 11.20

INHERITANCE EXAMPLE: ENGINE SIMULATION

This example expands on the engine simulation originally presented in chapter 10. Here the concepts of inheritance fuse with compositional design to yield a truly powerful combination.

ENGINE SIMULATION UML DIAGRAM

Figure 11-20 shows the UML diagram for this version of the engine simulation. Note now that most of the functionality of a part resides in the Part class. In addition to its constructor method, the Part class contains two private fields: `_partStatus` and `_partName`, and two public read-write properties: `Status`, `PartName`. It contains one read-only property named `IsWorking`, which simply returns true or false depending on the part's current status.

The `IManagedPart` interface declares two methods: `SetFault()` and `ClearFault()`. The `EnginePart` class extends the Part class and implements `IManagedPart`. The `EnginePart` class also contains one private field named `_registeredEngineNumber` and a corresponding public property named `RegisteredEngineNumber`. It has one read-only property named `PartIdentifier` that returns a string containing the name of the part and its registered engine number. It also defines a private utility method named `DisplayStatus()` that is called internally by the `SetFault()` and `ClearFault()` methods. The `EnginePart` class is declared to be an abstract class to prevent the creation of `EnginePart` objects with the `new` operator.

The classes `OilPump`, `FuelPump`, `Compressor`, `WaterPump`, `OxygenSensor`, and `TemperatureSensor` all extend from `EnginePart`. The `Engine` class is an aggregate of all of its parts. It contains an array of `EngineParts` named `_itsParts`. It also contains several other private fields, `_engineNumber` and `_isRunning`, along with their corresponding public properties `EngineNumber` and `IsRunning`. It has four public methods: `StartEngine()`, `StopEngine()`, `SetPartFault()`, and `ClearPartFault()`. It has one private method named `CheckEngine()`, which is used internally by the `StartEngine()`, `SetPartFault()`, and `ClearPartFault()` methods.

SIMULATION OPERATIONAL DESCRIPTION

Examples 11.31 and 11.32 give the source code for the `Engine` and `EngineTestApp` classes. Refer to them for this discussion. The results of running example 11.32 can be seen in figure 11-21.

```

Administrator: Command Prompt
C:\Projects\CFA\Chapter11\EngineSimulation>EngineTestApp
Part Created...
Compressor created...
Part Created...
FuelPump created...
Part Created...
OilPump created...
Part Created...
WaterPump created...
Part Created...
OxygenSensor created...
Part Created...
TemperatureSensor created...
Engine number 1 created
Checking engine number 1...
Engine number 1 working properly!
Engine number 1 started!
OilPump For Engine Number: 1 status is now: NOT_WORKING
The status of Engine number 1's OilPump is NOT_WORKING
Checking engine number 1...
OilPump For Engine Number: 1 NOT_WORKING
Engine number 1 malfunction!
Engine number 1 has been stopped!
OilPump For Engine Number: 1 status is now: WORKING
The status of Engine number 1's OilPump is WORKING
Checking engine number 1...
Engine number 1 working properly!
Checking engine number 1...
Engine number 1 working properly!
Engine number 1 started!
Engine number 1 has been stopped!
C:\Projects\CFA\Chapter11\EngineSimulation>

```

Figure 11-21: Results of Running the `EngineTestApp`

Referring first to example 11.32 — The `EngineTestApp` declares an `Engine` reference named `e1` and creates an `Engine` object with an engine number of 1. This is followed by a call to both the `StartEngine()` and `StopEngine()` methods. Next, a fault is set in the `OilPump` via a call to the `SetPartFault()` method. An attempt is then made to call `StartEngine()`, but because of the now faulty oil pump the engine will not start.

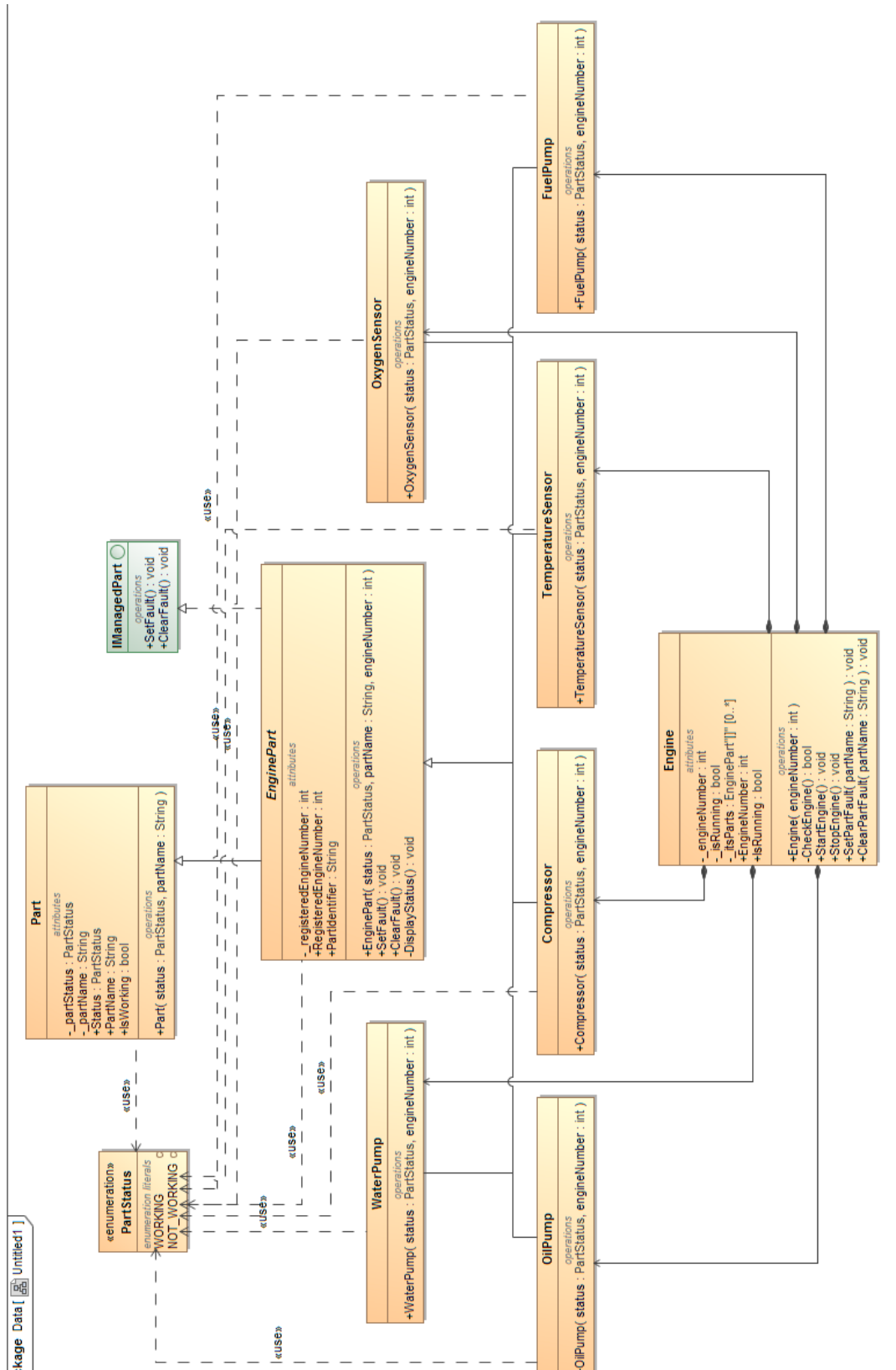


Figure 11-20: Engine Simulation UML Class Diagram

The fault is cleared with a call to `ClearPartFault()`, and the next call to `StartEngine()` works fine. You can follow this sequence of events in figure 11-21.

Compiling THE ENGINE SIMULATION CODE

You can compile the engine simulation code by putting all the code in one directory and issuing the following command: `csc *.cs`

COMPLETE ENGINE SIMULATION CODE LISTING

11.21 PartStatus.cs

```
1 namespace EngineSimulation {
2
3     public enum PartStatus { WORKING, NOT_WORKING }
4
5 }
```

11.22 Part.cs

```
1 using System;
2
3 namespace EngineSimulation {
4
5     public class Part {
6         private PartStatus _partStatus;
7         private String _partName;
8
9         public PartStatus Status {
10            get { return _partStatus; }
11            set { _partStatus = value; }
12        }
13
14        public String PartName {
15            get { return _partName; }
16            set { _partName = value; }
17        }
18
19        public bool IsWorking {
20            get { return (Status == PartStatus.WORKING); }
21        }
22
23        public Part(PartStatus status, String partName) {
24            _partName = partName;
25            _partStatus = status;
26            Console.WriteLine("Part Created...");
27        }
28    } // end class definition
29 } // end namespace
```

11.23 IManagedPart.cs

```
1 namespace EngineSimulation {
2     public interface IManagedPart {
3         void SetFault();
4         void ClearFault();
5     }
6 }
```

11.24 EnginePart.cs

```

1  using System;
2
3  namespace EngineSimulation {
4      public abstract class EnginePart : Part, IManagedPart {
5          private int _registeredEngineNumber;
6
7          public int RegisteredEngineNumber {
8              get { return _registeredEngineNumber; }
9              set { _registeredEngineNumber = value; }
10         }
11
12         public String PartIdentifier {
13             get { return PartName + " For Engine Number: " + _registeredEngineNumber; }
14         }
15
16         public EnginePart(PartStatus status, String partName, int engineNumber ):
17             base(status, partName){
18             _registeredEngineNumber = engineNumber;
19         }
20
21         public void SetFault() {
22             Status = PartStatus.NOT_WORKING;
23             DisplayStatus();
24         }
25
26         public void ClearFault() {
27             Status = PartStatus.WORKING;
28             DisplayStatus();
29         }
30
31         private void DisplayStatus(){
32             Console.WriteLine(PartIdentifier + " status is now: " + Status);
33         }
34     } // end class definition
35 } // end namespace

```

11.25 Compressor.cs

```

1  using System;
2
3  namespace EngineSimulation {
4
5      public class Compressor : EnginePart {
6
7          public Compressor(PartStatus status, int engineNumber):
8              base(status, "Compressor", engineNumber){
9              Console.WriteLine("Compressor created...");
10         }
11     } // end class
12 } // end namespace

```

11.26 FuelPump.cs

```

1  using System;
2
3  namespace EngineSimulation {
4
5      public class FuelPump : EnginePart {
6
7          public FuelPump(PartStatus status, int engineNumber):
8              base(status, "FuelPump", engineNumber){
9              Console.WriteLine("FuelPump created...");
10         }
11     } // end class
12 } // end namespace

```

11.27 OilPump.cs

```
1 using System;
2
3 namespace EngineSimulation {
4
5     public class OilPump : EnginePart {
6
7         public OilPump(PartStatus status, int engineNumber):
8             base(status, "OilPump", engineNumber){
9             Console.WriteLine("OilPump created...");
10        }
11    } // end class
12 } // end namespace
```

11.28 OxygenSensor.cs

```
1 using System;
2
3 namespace EngineSimulation {
4
5     public class OxygenSensor : EnginePart {
6
7         public OxygenSensor(PartStatus status, int engineNumber):
8             base(status, "OxygenSensor", engineNumber){
9             Console.WriteLine("OxygenSensor created...");
10        }
11    } // end class
12 } // end namespace
```

11.29 TemperatureSensor.cs

```
1 using System;
2
3 namespace EngineSimulation {
4
5     public class TemperatureSensor : EnginePart {
6
7         public TemperatureSensor(PartStatus status, int engineNumber):
8             base(status, "TemperatureSensor", engineNumber){
9             Console.WriteLine("TemperatureSensor created...");
10        }
11    } // end class
12 } // end namespace
```

11.30 WaterPump.cs

```
1 using System;
2
3 namespace EngineSimulation {
4
5     public class WaterPump : EnginePart {
6
7         public WaterPump(PartStatus status, int engineNumber):
8             base(status, "WaterPump", engineNumber){
9             Console.WriteLine("WaterPump created...");
10        }
11    } // end class
12 } // end namespace
```

11.31 Engine.cs

```
1 using System;
2
3 namespace EngineSimulation {
4     public class Engine {
5         private int _engineNumber;
6         private bool _isRunning;
7         private EnginePart[] _itsParts;
```



```

8
9     public int EngineNumber {
10         get { return _engineNumber; }
11     }
12
13     public bool IsRunning {
14         get { return _isRunning; }
15     }
16
17     public Engine(int engineNumber){
18         _engineNumber = engineNumber;
19         _isRunning = false;
20         _itsParts = new EnginePart[6];
21         _itsParts[0] = new Compressor(PartStatus.WORKING, EngineNumber);
22         _itsParts[1] = new FuelPump(PartStatus.WORKING, _engineNumber);
23         _itsParts[2] = new OilPump(PartStatus.WORKING, _engineNumber);
24         _itsParts[3] = new WaterPump(PartStatus.WORKING, _engineNumber);
25         _itsParts[4] = new OxygenSensor(PartStatus.WORKING, _engineNumber);
26         _itsParts[5] = new TemperatureSensor(PartStatus.WORKING, _engineNumber);
27         Console.WriteLine("Engine number {0} created", _engineNumber);
28     }
29
30     private bool CheckEngine(){
31         Console.WriteLine("Checking engine number {0}...", _engineNumber);
32         bool is_working = false;
33
34         for(int i=0; i<_itsParts.Length; i++){
35             is_working = _itsParts[i].IsWorking;
36             if(!is_working){
37                 Console.WriteLine(_itsParts[i].PartIdentifier + " " + _itsParts[i].Status);
38                 break;
39             }
40         }
41
42         if(is_working){
43             Console.WriteLine("Engine number {0} working properly!", _engineNumber);
44         }else{
45             Console.WriteLine("Engine number {0} malfunction!", _engineNumber);
46             StopEngine();
47         }
48         return is_working;
49     }
50
51     public void StartEngine(){
52         if(!_isRunning){
53             _isRunning = CheckEngine();
54         }if(!_isRunning){
55             Console.WriteLine("Engine number {0} failed to start!", _engineNumber);
56         }else{
57             Console.WriteLine("Engine number {0} started!", _engineNumber);
58         }
59     }else{
60         Console.WriteLine("Engine number {0} is already running!", _engineNumber);
61     }
62 }
63
64     public void StopEngine(){
65         if(_isRunning){
66             _isRunning = false;
67             Console.WriteLine("Engine number {0} has been stopped!", _engineNumber);
68         }else{
69             Console.WriteLine("Engine number {0} is not running!", _engineNumber);
70         }
71     }
72

```

```

73     public void SetPartFault(String partName){
74         for(int i=0; i<_itsParts.Length; i++){
75             if(_itsParts[i].PartName.Equals(partName)){
76                 _itsParts[i].SetFault();
77                 Console.WriteLine("The status of Engine number {0}'s {1} is {2}", _engineNumber,
78                                 _itsParts[i].PartName, _itsParts[i].Status);
79                 break;
80             }
81         }
82         CheckEngine();
83     }
84
85     public void ClearPartFault(String partName){
86         for(int i=0; i<_itsParts.Length; i++){
87             if(_itsParts[i].PartName.Equals(partName)){
88                 _itsParts[i].ClearFault();
89                 Console.WriteLine("The status of Engine number {0}'s {1} is {2}", _engineNumber,
90                                 _itsParts[i].PartName, _itsParts[i].Status);
91                 break;
92             }
93         }
94         CheckEngine();
95     }
96 } // end class
97 } // end namespace

```

11.32 EngineTestApp.cs

```

1  using EngineSimulation;
2
3  public class EngineTestApp {
4      public static void Main(){
5          Engine e1 = new Engine(1);
6          e1.StartEngine();
7          e1.SetPartFault("OilPump");
8          e1.ClearPartFault("OilPump");
9          e1.StartEngine();
10         e1.StopEngine();
11     } // end Main()
12 } // end class

```

Polymorphic Engine Simulation (MEF)

In this section, I want to rework the Managed Extensibility Framework (MEF) version of the Engine Simulation presented in chapter 10 to use interfaces and abstract base classes. This enables the use of parts polymorphically, which leads to cleaner code, less redundancy, and increased application architectural flexibility.

As you have learned in this chapter and saw in the previous section, targeting an interface enables you to treat different types of objects as if they were all the same type. Though I don't formally introduce you to the .NET Collections framework until Chapter 14 — Collections, I will, in this example, use a generic collection — `List<EnginePart>` (...read, "List that can contain EnginePart objects.") — to hold references to an engine's part objects for quick access. I'll explain more when we discuss that section of code.

I have, for the most part, kept the same inheritance hierarchy I employed in the previous example. The code is updated to reflect modern C# idioms regarding the use of auto-properties, tuples, tuple unpacking, and named tuple fields, which was introduced in C# 7.

Regarding further differences you'll see between the MEF version presented in chapter 10 and the present example, this one is slightly more conceptually complex. Increased conceptual complexity introduces a corresponding increase in physical complexity, meaning there are now more source files spread

across multiple projects. I am using Visual Studio 2017 to manage the physical complexity and to build and run the example.

Solution: PolymorphicEngineSimulationMEF

The PolymorphicEngineSimulationMEF solution contains four projects: *Common*, *EngineParts*, *Engines*, and *EngineTester*. The projects *Common*, *EngineParts*, and *Engines* all produce class libraries (DLLs). The *EngineTester* project is a console application.

PROJECT: COMMON (TARGET: .NET FRAMEWORK 4.7.2, OUTPUT TYPE: CLASS LIBRARY)

The Common project contains the following source files: *PartStatusEnum.cs*, *IPart.cs*, *Part.cs*, *IManagedPart.cs*, *IEngine.cs*, *EnginePart.cs*. One thing to note about the classes and interfaces contained within the Common project is the absence of MEF Imports or Exports.

11.33 *PartStatusEnum.cs*

```
1 namespace Common {
2     public enum PartStatus { NOT_WORKING, WORKING }
3 }
```

Referring to example 11.33 — The *PartStatus* enumeration remains unchanged from chapter 10.

11.34 *IPart.cs*

```
1 namespace Common {
2     public interface IPart {
3
4         string Name {
5             get;
6             set;
7         }
8
9         PartStatus Status {
10            get;
11            set;
12        }
13
14        bool IsWorking {
15            get;
16            set;
17        }
18    }
19 }
```

Referring to example 11.34 — The *IPart* interface provides a specification for a part. In this case, a part has three public properties: *Name*, *Status*, and *IsWorking*.

11.35 *Part.cs*

```
1 namespace Common {
2     public abstract class Part : IPart {
3
4         public string Name {
5             get;
6             set;
7         }
8
9         public PartStatus Status {
10            get;
11            set;
12        } = PartStatus.WORKING;
13
14        public bool IsWorking {
```

```

15     get {
16         return Status == PartStatus.WORKING ? true : false;
17     }
18     set {
19         if (value) Status = PartStatus.WORKING;
20         else Status = PartStatus.NOT_WORKING;
21     }
22 }
23
24 public Part(string name) {
25     Name = name;
26 }
27
28 public override string ToString() {
29     return Name;
30 }
31
32 } // end class
33 } // end namespace

```

Referring to example 11.35 — The abstract `Part` class implements `IPart` and provides an implementation for all three properties specified by the interface. It also provides a constructor, and overrides the `Object.ToString()` method. Choosing to make the `Part` class abstract is a design decision. Even though it provides an implementation for all properties specified by `IPart`, I want to prevent the creation of plain ol' parts. You'll see why here in a second.

11.36 IManagedPart.cs

```

1 namespace Common {
2     internal interface IManagedPart {
3         void SetFault();
4         void ClearFault();
5     }
6 }

```

Referring to example 11.36 — The `IManagedPart` interface specifies two methods: `SetFault()` and `ClearFault()`. I could have included these methods in the `IPart` interface but that little, nagging design voice inside my head said it might come in handy to have a separate interface to express the notion of a part that can have a fault set and cleared, so here it is. Application architectures often spring forth and evolve in this fashion, for better or for worse.

11.37 ManagedPart.cs

```

1 using System;
2
3 namespace Common {
4
5     public abstract class EnginePart : Part, IManagedPart {
6
7         public IEngine RegisteredEngine {
8             get;
9             set;
10        }
11
12        public string PartIdentifier {
13            get {
14                return Name + " for Engine No. " + RegisteredEngine.EngineNumber;
15            }
16        }
17
18        public EnginePart(string name) : base(name) {
19
20        }
21
22        public void SetFault() {

```

```

23     IsWorking = false;
24     DisplayStatus();
25     RegisteredEngine.StopEngine();
26
27 }
28
29 public void ClearFault() {
30     IsWorking = true;
31     DisplayStatus();
32 }
33
34 private void DisplayStatus() {
35     Console.WriteLine(PartIdentifier + " is now " + Status);
36 }
37
38 } // end class
39 } // end namespace

```

Referring to example 11.37 — The abstract class `EnginePart` extends `Part` and implements `IManagedPart`. It adds a read-write property named `RegisteredEngine`. The idea here is that `EnginePart` objects will have a reference to their containing `Engine` object, so that when a fault is injected via the `SetFault()` method, the engine is shut down via a call to `RegisteredEngine.StopEngine()`.

11.38 *IEngine.cs*

```

1 namespace Common {
2     public interface IEngine {
3         int EngineNumber {
4             get;
5             set;
6         }
7
8         bool IsRunning {
9             get;
10            set;
11        }
12
13        bool IsWorking {
14            get;
15        }
16
17        (bool IsWorking, string[] StatusMessages) CheckEngine();
18        void StartEngine();
19        void StopEngine();
20    }
21 }

```

Referring to example 11.38 — The `IEngine` interface provides a specification for what it means to be an `Engine`. In this case, an `Engine` has three properties: `EngineNumber`, `IsRunning`, and `IsWorking`, and three methods: `CheckEngine`, which, for convenience, returns a two-item tuple with named fields. (C# 7), `StartEngine()`, and `StopEngine()`.

PROJECT: ENGINEPARTS (TARGET: .NET FRAMEWORK 4.7.2, OUTPUT TYPE: CLASS LIBRARY)

The `EngineParts` project contains the following source files: `OilPump.cs`, `FuelPump.cs`, `Compressor.cs`, `OxygenSensor.cs`, `TemperatureSensor.cs`.

One big difference you'll notice between the concrete part classes listed here and their chapter 10 cousins is that all the heavy lifting of what it means to be an `EnginePart` is performed by the `Part` and `EnginePart` abstract classes found in the `Common` project, so these class listings have much less code repetition.

11.39 OilPump.cs

```

1  using Common;
2  using System;
3  using System.ComponentModel.Composition;
4
5  namespace EngineParts {
6
7      [Export(typeof(OilPump))]
8      public class OilPump : EnginePart {
9
10         [ImportingConstructor]
11         public OilPump() : base("OilPump") {
12             Console.WriteLine(this.GetType().Name + " Created!");
13         }
14     }
15 }

```

Referring to example 11.39 — The `OilPump` class exports type `OilPump` and applies the `[ImportingConstructor]` attribute to its constructor. That's it, at least for this example. The abstract classes `Part` and `EnginePart` supply all the functionality. The remaining engine parts need no further explanation, so I'll just list the code.

11.40 FuelPump.cs

```

1  using Common;
2  using System;
3  using System.ComponentModel.Composition;
4
5  namespace EngineParts {
6
7      [Export(typeof(FuelPump))]
8      public class FuelPump : EnginePart {
9
10         [ImportingConstructor]
11         public FuelPump() : base("FuelPump") {
12             Console.WriteLine(this.GetType().Name + " Created!");
13         }
14     }
15 }

```

11.41 Compressor.cs

```

1  using Common;
2  using System;
3  using System.ComponentModel.Composition;
4
5  namespace EngineParts {
6
7      [Export(typeof(Compressor))]
8      public class Compressor : EnginePart {
9
10         [ImportingConstructor]
11         public Compressor() : base("Compressor") {
12             Console.WriteLine(this.GetType().Name + " Created!");
13         }
14     }
15 }

```

11.42 OxygenSensor.cs

```

1  using Common;
2  using System;
3  using System.ComponentModel.Composition;
4
5  namespace EngineParts {
6
7      [Export(typeof(OxygenSensor))]

```

```

8     public class OxygenSensor : EnginePart {
9
10        [ImportingConstructor]
11        public OxygenSensor() : base("OxygenSensor") {
12            Console.WriteLine(this.GetType().Name + " Created!");
13        }
14    }
15 }

```

11.43 TemperatureSensor.cs

```

1     using Common;
2     using System;
3     using System.ComponentModel.Composition;
4
5     namespace EngineParts {
6
7         [Export(typeof(TemperatureSensor))]
8         public class TemperatureSensor : EnginePart {
9
10            [ImportingConstructor]
11            public TemperatureSensor() : base("TemperatureSensor") {
12                Console.WriteLine(this.GetType().Name + " Created!");
13            }
14        }
15    }

```

PROJECT: ENGINES (TARGET: .NET FRAMEWORK 4.7.2, OUTPUT TYPE: CLASS LIBRARY)

The Engines project contains one class named Engine. When compared with the MEF Engine class given in chapter 10, this version is much cleaner though structured differently. The primary and significant difference here is the use of polymorphism. This is made possible through the use of abstract classes and interfaces. For the parts, I'm targeting the EnginePart abstract class. Another difference is the application of the [Import] attribute to properties vs. fields.

11.44 Engine.cs

```

1     using Common;
2     using EngineParts;
3     using System;
4     using System.Collections.Generic;
5     using System.ComponentModel.Composition;
6     using System.ComponentModel.Composition.Hosting;
7     using System.Reflection;
8     using System.Text;
9
10
11    namespace Engines {
12
13        [Export]
14        public class Engine : IEngine {
15
16            /***** Part Properties *****/
17            [Import(typeof(OilPump))]
18            public EnginePart OilPump {
19                get;
20                set;
21            }
22
23            [Import(typeof(FuelPump))]
24            public EnginePart FuelPump {
25                get;
26                set;
27            }

```

```

28
29     [Import(typeof(Compressor))]
30     public EnginePart Compressor {
31         get;
32         set;
33     }
34
35     [Import(typeof(TemperatureSensor))]
36     public EnginePart TemperatureSensor {
37         get;
38         set;
39     }
40
41     [Import(typeof(OxygenSensor))]
42     public EnginePart OxygenSensor {
43         get;
44         set;
45     }
46
47     /***** Public Properties *****/
48     [Export]
49     public int EngineNumber {
50         get;
51         set;
52     }
53
54     public bool IsRunning {
55         get;
56         set;
57     } = false;
58
59     public bool IsWorking {
60         get {
61             return CheckEngine().IsWorking;
62         }
63     }
64
65     public int OilPumpEngineNumber {
66         get { return OilPump.RegisteredEngine.EngineNumber; }
67     }
68
69     public int FuelPumpEngineNumber {
70         get { return FuelPump.RegisteredEngine.EngineNumber; }
71     }
72
73     public int CompressorEngineNumber {
74         get { return Compressor.RegisteredEngine.EngineNumber; }
75     }
76
77     public int TemperatureSensorEngineNumber {
78         get { return TemperatureSensor.RegisteredEngine.EngineNumber; }
79     }
80
81     public int OxygenSensorEngineNumber {
82         get { return OxygenSensor.RegisteredEngine.EngineNumber; }
83     }
84
85     /***** MEF Parts Container *****/
86     private CompositionContainer _container;
87
88     /***** List of EngineParts *****/
89     private List<EnginePart> _parts;
90
91     /***** Constructor *****/
92     [ImportingConstructor]

```



```

93     public Engine(int engine_number) {
94         EngineNumber = engine_number;
95         try {
96             // Check runtime directory for DLLs that contain parts with matching imports
97             var catalog = new DirectoryCatalog(".");
98             // Create the catalog
99             _container = new CompositionContainer(catalog);
100            this._container.ComposeParts(this);
101            _parts = GetParts();
102            foreach (EnginePart part in _parts) {
103                part.RegisteredEngine = this;
104            }
105        } catch (Exception e) {
106            Console.WriteLine(e);
107        }
108    } // end constructor
109
110
111    public (bool IsWorking, string[] StatusMessages) CheckEngine() {
112        Console.WriteLine("CheckEngine() method called...");
113        StringBuilder status_messages = new StringBuilder();
114        bool working = true;
115        // Check each engine property
116        foreach (EnginePart part in _parts) {
117            if (part.IsWorking) {
118                status_messages.Append(part + " -> working!");
119            } else {
120                status_messages.Append(part + " -> faulty!");
121                working = false;
122            }
123        }
124        return (working, status_messages.ToString().Split(','));
125    }
126
127
128    public void StartEngine() {
129        Console.WriteLine("Checking Engine No. {0} ", EngineNumber);
130        if (!IsRunning) {
131            if (IsWorking) {
132                Console.WriteLine("Starting Engine No. {0} ", EngineNumber);
133                IsRunning = true;
134                Console.WriteLine("Engine No. {0} is running!", EngineNumber);
135            } else {
136                Console.WriteLine("Engine No. {0} has a problem...", EngineNumber);
137                foreach (string s in CheckEngine().StatusMessages) {
138                    Console.WriteLine(s);
139                }
140            }
141        } else {
142            Console.WriteLine("Engine No. {0} is already running!", EngineNumber);
143        }
144    } // end StartEngine()
145
146
147    public void StopEngine() {
148        Console.WriteLine("Stopping Engine No. {0} ", EngineNumber);
149        IsRunning = false;
150        Console.WriteLine("Engine No. {0} is shut down.", EngineNumber);
151    }
152
153    private List<EnginePart> GetParts() {
154        List<EnginePart> parts = new List<EnginePart>();
155        foreach (PropertyInfo prop in this.GetType().GetProperties()) {
156            if (prop.PropertyType == typeof(EnginePart)) {
157                parts.Add((EnginePart)prop.GetValue(this));

```

```

158     }
159     }
160     return parts;
161 }
162
163 } // end class
164 } // end namespace

```

Referring to example 11.44 — Let’s start from the bottom this time. Beginning on line 153, the `GetParts()` method uses reflection (`System.Reflection`) to step through each `Engine` property, pick out the ones of type `EnginePart`, and add them to the parts list and then return this list. The `GetParts()` method is used in the `Engine` constructor on line 101 to initialize `Engine`’s `_parts` field. The reason for doing this is to provide a convenient way to access engine parts after they’ve been automatically imported and manipulate them polymorphically.

The `CheckEngine()` method on line 111 steps through each part in the `_parts` collection and determines if the engine is working or not by checking each part’s `IsWorking` property and adding the appropriate message to `status_messages` and setting `working` to false if one or more of the parts are not working. The `CheckEngine()` method returns a tuple with named item fields. Named item fields are a new feature of the C# language version 7. The use of named tuple items really cleans up `Engine`’s `IsWorking` property and allows us to just make a call to `CheckEngine().IsWorking` as is shown beginning on line 59.

The `[Import]` attributes applied to each `Engine` part property explicitly specify the type of part that must be imported to satisfy the import. This is necessary because now all the `EnginePart` properties have the same type — `EnginePart` — but we don’t want just any ol’ `EnginePart` to fill the shoes of a `FuelPump`. No, we want a `FuelPump`, and one that’s also an `EnginePart`.

PROJECT: ENGINETESTER (TARGET: .NET FRAMEWORK 4.7.2, OUTPUT TYPE: CONSOLE APPLICATION)

The `EngineTester` project contains one class, `MainApp`, which takes advantage of the new `IManagedPart` interface methods `SetFault()` and `ClearFault()`.

11.45 *MainApp.cs*

```

1  using Engines;
2  using System;
3
4
5  namespace EngineTester {
6      public class MainApp {
7          private static void Main(string[] args) {
8              Console.WriteLine("Instantiating Engine Object...");
9              Engine e1 = new Engine(1);
10             Console.WriteLine("-----");
11             Console.WriteLine("Reading Part Engine Numbers...");
12             Console.WriteLine("OilPump Engine Number: {0}", e1.OilPumpEngineNumber);
13             Console.WriteLine("FuelPump Engine Number: {0}", e1.FuelPumpEngineNumber);
14             Console.WriteLine("Compressor Engine Number: {0}", e1.CompressorEngineNumber);
15             Console.WriteLine("TemperatureSensor Engine Number: {0}",
16                 e1.TemperatureSensorEngineNumber);
17             Console.WriteLine("OxygenSensor Engine Number: {0}", e1.OxygenSensorEngineNumber);
18             Console.WriteLine("-----");
19
20             Console.WriteLine("\nPress Enter to Start Engine Number: {0}", e1.EngineNumber);
21             Console.ReadKey();
22
23             e1.StartEngine();
24
25             Console.WriteLine("\nPress Enter to Set FuelPump Fault...");
26             Console.ReadKey();
27
28             e1.FuelPump.SetFault();

```

```

29
30     Console.WriteLine("\nPress Enter to Try to Start Engine Number: "
31         + "{0} with Faulty FuelPump", e1.EngineNumber);
32     Console.ReadKey();
33
34     e1.StartEngine();
35
36     Console.WriteLine("\nPress Enter to Fix FuelPump and Restart Engine...");
37     Console.ReadKey();
38
39     e1.FuelPump.ClearFault();
40     e1.StartEngine();
41
42     Console.WriteLine("\nPress Enter to Stop Engine...");
43     Console.ReadKey();
44
45     e1.StopEngine();
46
47     Console.WriteLine("\n\nPress Enter to Exit the Engine Demo...");
48     Console.ReadKey();
49 }
50 }
51 }

```

Referring to example 11.45 — I've just highlighted lines 28 and 39 to illustrate the use of the `Set-Fault()` and `ClearFault()`. The Engine's `FuelPump` is accessed via the corresponding property. The `FuelPump` property type is `EnginePart`, and `EnginePart` implements `IManagedPart`. The rest of the example remains unchanged from chapter 10.

Quick Review

The Polymorphic Engine Simulation (MEF) demonstrates the use of interfaces and abstract base classes to achieve polymorphic behavior along with automatic object composition using the Managed Extensibility Framework, a powerful combination. `[Import]` attributes can be applied to properties as well as fields. In the case where properties or fields share a common type for polymorphic purposes, but only a particular type of object can fulfill the import, the `[Import]` attribute must explicitly state the import type.

Beginning with C# 7, tuples can have named item fields. This significantly aides code readability and reduces mistakes caused by accessing the wrong tuple item.

SUMMARY

Inheritance serves three essential purposes: 1) it is an object-oriented design mechanism that enables you to think and reason about a program's structure in terms of *generalized* and *specialized* class behavior, 2) it provides you with a measure of code reuse within your program by locating common class behavior in base classes, and 3) it provides a means to incrementally develop your programs over time.

Classes that belong to an inheritance hierarchy participate in an *is a* relationship between themselves and their chain of base classes. This *is a* relationship is transitive in the direction of specialized to generalized classes but not vice versa.

Class and interface constructs introduce new, user-defined data types. The interface construct is used to specify a set of authorized type operations but omits their behavior; the class construct is used to specify a set of authorized type operations and, optionally, their behavior as well. A class construct, like an interface, can omit the bodies of one or more of its members, however, such members must be declared to be abstract. A class that declares one or more of its members to be abstract must itself be declared to be an *abstract class*. Abstract class objects cannot be created with the `new` operator.

```

C:\Projects\CSharpForArtists3rdEdition\Chapter11\PolymorphicEngineSim...
Instantiating Engine Object...
OilPump Created!
FuelPump Created!
Compressor Created!
TemperatureSensor Created!
OxygenSensor Created!
-----
Reading Part Engine Numbers...
OilPump Engine Number: 1
FuelPump Engine Number: 1
Compressor Engine Number: 1
TemperatureSensor Engine Number: 1
OxygenSensor Engine Number: 1
-----
Press Enter to Start Engine Number: 1
Checking Engine No. 1
CheckEngine() method called...
Starting Engine No. 1
Engine No. 1 is running!

Press Enter to Set FuelPump Fault...
FuelPump for Engine No. 1 is now NOT_WORKING
Stopping Engine No. 1
Engine No. 1 is shut down.

Press Enter to Try to Start Engine Number: 1 with Faulty FuelPump
Checking Engine No. 1
CheckEngine() method called...
Engine No. 1 has a problem...
CheckEngine() method called...
OilPump -> working!
FuelPump -> faulty!
Compressor -> working!
TemperatureSensor -> working!
OxygenSensor -> working!

Press Enter to Fix FuelPump and Restart Engine...
FuelPump for Engine No. 1 is now WORKING
Checking Engine No. 1
CheckEngine() method called...
Starting Engine No. 1
Engine No. 1 is running!

Press Enter to Stop Engine...
Stopping Engine No. 1
Engine No. 1 is shut down.

Press Enter to Exit the Engine Demo...

```

Figure 11-22: Results of Running Example 11.45

A *base class* implements default behavior in the form of public, protected, internal, and protected internal members that can be inherited by *derived classes*. There are three reference/object combinations: 1) if the base class is a *concrete class*, meaning it is not abstract, then a base class reference can point to a base class object, 2) a base class reference can point to a derived class object, and 3) a derived class reference can point to a derived class object.

Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without *casting* as long as the reference's type supports the method you are trying to call. You can force, or coerce, the compiler to treat a reference to an object of one type as if it were a reference to an object of another. This is extremely helpful in some circumstances, but as a rule, use casting sparingly. Also, casting only works if the object really is of the type you are casting it to.

Derived classes can *override* base class behavior by providing *overriding methods*. An *overriding method* is a method in a derived class that has the same method signature as the base class method it is overriding. Use the `virtual` keyword to declare an overrideable base class method. Use the `override` keyword to define an overriding derived class method. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

An *abstract member* is a member that omits its body and has no implementation behavior. A class that declares one or more abstract members must be declared to be abstract. The primary purpose of an *abstract class* is to provide a specification for behavior whose implementation is expected to be found in some derived class further down the inheritance hierarchy. Designers employ abstract classes to provide a measure of application architectural stability.

The purpose of an interface is to specify behavior. Interfaces can have four types of members: 1) properties, 2) methods, 3) events, and 4) indexers. Classes can inherit from or extend only one other class, but they can implement as many interfaces as are required. Interfaces can extend as many other interfaces as necessary.

Use the access modifiers `private`, `protected`, `public`, `internal`, and `protected internal` to control horizontal and vertical member access.

Use the `sealed` keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

Polymorphic behavior is achieved in a program by targeting a set of operations specified by a base class or interface and manipulating their derived class objects via those operations. This uniform treatment of derived class objects results in cleaner code that's easier to extend and maintain. Polymorphic behavior is the essence of object-oriented programming.

The Polymorphic Engine Simulation (MEF) demonstrates the use of interfaces and abstract base classes to achieve polymorphic behavior along with automatic object composition using the Managed Extensibility Framework, a powerful combination. `[Import]` attributes can be applied to properties as well as fields. In the case where properties or fields share a common type for polymorphic purposes, but only a particular type of object can fulfill the import, the `[Import]` attribute must explicitly state the import type.

Beginning with C# 7, tuples can have named item fields. This significantly aides code readability and reduces mistakes caused by accessing the wrong tuple item.

Skill-Building Exercises

- Simple Inheritance:** Write a small program to test the effects of inheritance. Create a class named `ClassA` that implements the following methods: `A()`, `B()`, and `C()`. Each method should print a short text message to the console. Create a default constructor for `ClassA` that prints a message to the console announcing the creation of a `ClassA` object. Next, create a class named `ClassB` that extends `ClassA`. Give `ClassB` a default constructor that announces the creation of a `ClassB` object. In a test driver program create three references. Two of the references should be of type `ClassA` and the third should be of type `ClassB`. Initialize the first reference to point to a `ClassA` object, initialize the second reference to point to a `ClassB` object, and initialize the third reference to point to a `ClassB` object as well. Call the methods `A()`, `B()`, and `C()` via each of the references. Run the test driver program and note the results.
- Overriding Methods:** Reusing some of the code you created in the previous exercise create another class named `ClassC` that extends `ClassA` and provides overriding methods for each of `ClassA`'s methods `A()`, `B()`, and `C()`. Have each of the methods defined in `ClassC` print short messages to the console. In the test driver program declare three references, the first two of type `ClassA` and the third of type `ClassC`. Initialize the first reference to point to an object of type `ClassA`, the second to point to an object of type `ClassC`, and the third to point to an object of `ClassC` as well. Call the methods `A()`, `B()`, and `C()` via each of the references. Run the test driver program and note the results.
- Abstract Classes:** Create an abstract class named `AbstractClassA` and give it a default constructor and three abstract methods named `A()`, `B()`, and `C()`. Create another class named `ClassB` that extends `ClassA`.

Provide overriding methods for each of the abstract methods declared in ClassA. Each overriding method should print a short text message to the console. Create a test driver program that declares two references. The first reference should be of type ClassA, the second reference should be of type ClassB. Initialize the first reference to point to an object of type ClassB, and the second reference to point to an object of ClassB as well. Call the methods A(), B(), and C() via each of the references. Run the program and note the results.

4. **Interfaces:** Convert the abstract class you created in the previous exercise to an interface. What changes did you have to make to the code? Compile your interface and test driver program code, re-run the program and note the results.
5. **Mental Exercise:** Consider the following scenario: Given an abstract base class named ClassOne with the following abstract public interface methods A(), B(), and C(). Given a class named ClassTwo that derives from ClassOne, provides implementations for each of ClassOne's abstract methods, and defines one additional method named D(). Now, you have two references. One is of type ClassOne, the other of type ClassTwo.

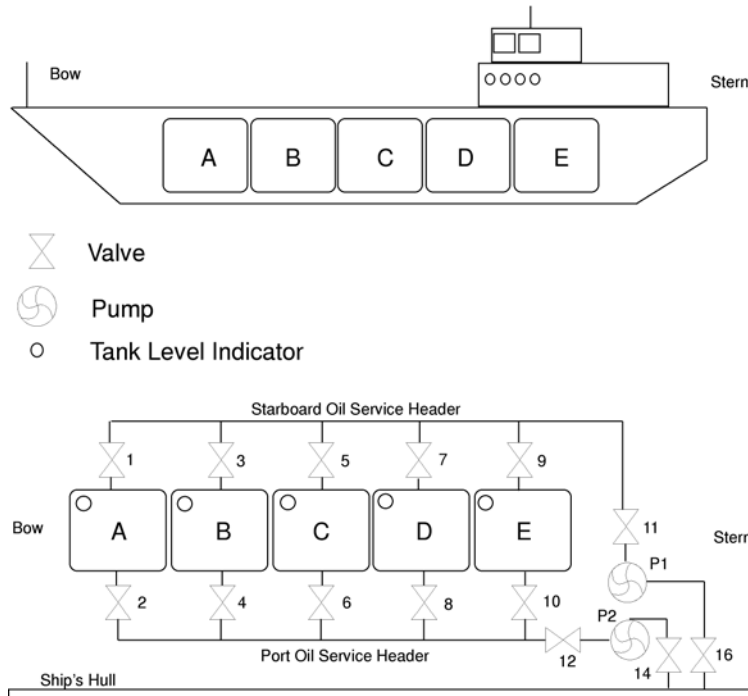
Answer these questions: What methods can be called via the ClassOne reference without casting? Likewise, what methods can be called via the ClassTwo reference without casting?

6. **Research The Managed Extensibility Framework (MEF):** Consult Microsoft's documentation about the Managed Extensibility Framework. Answer the following questions: What problem is the MEF trying to solve? What's the purpose of an Export? What's the purpose of an Import? How does the use of abstract classes of interfaces affect Exports and Imports? What is meant by the Attributed Programming Model?

SUGGESTED PROJECTS

1. **Draw Sequence Diagram:** Draw a UML sequence diagram of the Engine constructor call. Refer to the code supplied in examples 11.21 through 11.32.
2. **Draw Sequence Diagram:** Draw a UML sequence diagram of the Engine StartEngine() method.
3. **Draw Sequence Diagram:** Draw a UML sequence diagram of the Engine CheckEngine() method.
4. **Extend Functionality:** Extend the functionality of the Employee example given in this chapter. Create a subclass named PartTimeEmployee that extends HourlyEmployee. Limit the number of hours a PartTimeEmployee can have to 30 hours per pay period.
5. **Oil Tanker Pumping System:** Design and create an oil tanker pumping system simulation. Assume your tanker ship has five oil cargo compartments as shown in the diagram below.

Each compartment can be filled and drained from either the port or starboard service header. The oil pumping system consists of 14 valves numbered 1 through 16. Even-numbered valves are located on the port side of the ship and odd-numbered valves are located on the starboard side of the ship. **Note:** Valve numbers 13 and 15 are not used.



The system also consists of two pumps that can be run in two speeds, slow or fast speed, and in two directions, drain and fill. When a pump is running in the drain direction it is taking a suction from the tank side. When running in the fill direction it is taking a suction from the hull side. Assume a pumping capacity of 1,000 gallons per minute in fast mode.

Each tank contains one tank-level indicator that is a type of sensor. The indicators sense a continuous tank level from 0 (empty) to 100,000 gallons.

Your program should let you drain and fill the oil compartments by opening and closing valves and starting and setting pump speeds. For instance, to fill tank A quickly you could open valves 1, 2, 11, 12, 14 and 16, and start pumps P1 and P2 in the fill direction in the fast mode.

6. Oil Tanker Pumping System (MEF): Implement the Oil Tanker Pumping System project given in Suggested Project #5 using the Managed Extensibility Framework.

Self-Test Questions

1. What are the three essential purposes of inheritance?
2. A class that belongs to an inheritance hierarchy participates in what type of relationship with its base class?
3. Describe the relationship between the terms interface, class, and type.
4. How do you express generalization and specialization in a UML class diagram? You may draw a picture to answer the question.
5. Describe how to override a base class method in a derived class.

6. Why would it be desirable to override a base class method in a derived class?
7. What's the difference between an ordinary method and an abstract method?
8. How many abstract methods must a class have before it must be declared to be an abstract class?
9. List several differences between classes and interfaces.
10. How do you express an abstract class in a UML class diagram?
11. Hi, I'm a class that declares a set of interface methods but fails to provide an implementation for those methods. What type of class am I?
12. List the four authorized members of an interface.
13. Which two ways can you express realization in a UML class diagram? You may use pictures to answer the question.
14. How do you call a base class constructor from a derived class constructor?
15. How do you call a base class method, other than a constructor, from a derived class method?
16. Describe the effects of using the access modifiers `public`, `private`, `protected`, `internal`, and `protected internal` has on horizontal and vertical member access.
17. What can you do to prevent or stop a class from being inherited?
18. What can you do to prevent a method from being overridden in a derived class?
19. State, in your own words, a good definition for the term *polymorphism*.
20. What MEF attribute would you use to indicate an export?
21. What MEF attribute would you use to indicate an import?

REFERENCES

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-57168-4

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN: 0-8053-5340-2

Babak Sadr. *Unified Objects: Object-Oriented Programming Using C++*. The IEEE Computer Society, Los Alamitos, CA. ISBN: 0-8186-7733-3

Antero Taivalsaari. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, Septem-

ber 1996, pp. 438 - 479.

Clyde Ruby and Gary T. Levens. *Safely Creating Correct Subclasses without Seeing Superclass Code*. In OOPSLA '00 Conference Proceedings.

Derek Rayside and Gerard T. Campbell. *An Aristotelian Understanding of Object-Oriented Programming*. OOPSLA '00 Conference Proceedings.

ECMA-335 Common Language Infrastructure (CLI), 6th Edition, June 2012 <http://www.ecma-international.org/publications/standards/Ecma-335.htm>

ECMA-334 C# Language Specification, 4th Edition, June 2006 <http://www.ecma-international.org/publications/standards/Ecma-334.htm>

Microsoft Developer Network (MSDN) <http://www.msdn.com>

Microsoft Documentation, Managed Extensibility Framework (MEF) <https://docs.microsoft.com/en-us/dotnet/framework/mef/>

NOTES
