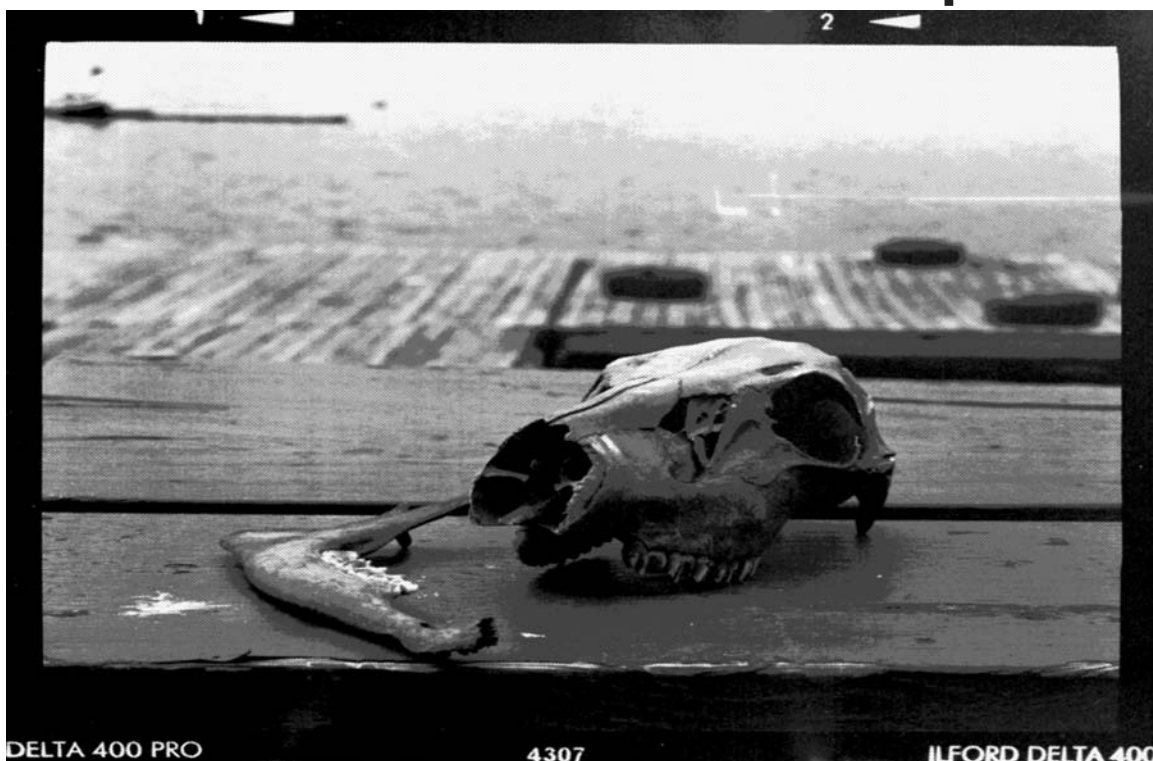


CHAPTER 9

Pentax 67 / SMC Takumar 150/2.8 / Ilford Delta 400



DEER Skull

TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

LEARNING OBJECTIVES

- *Describe the purpose and use of abstract data types*
- *State the purpose and use of an enumeration*
- *State the purpose and use of a structure*
- *State the purpose and use of a class*
- *List and describe the differences between structures and classes*
- *State the purpose and use of methods*
- *Demonstrate your ability to create structures, classes, and methods*
- *State the purpose and use of overloaded methods*
- *State the purpose and use of constructor methods*
- *State the definition of the term “method signature”*
- *Demonstrate your ability to overload ordinary methods and constructor methods*
- *State the purpose and use of the keyword “static”*
- *Demonstrate your ability to create class fields and methods*
- *State the purpose and use of a UML class diagram*
- *Demonstrate your ability to create classes that represent abstract data types*

INTRODUCTION

A computer program is a model of a real world problem. But real world problems are notoriously complex. It is impossible to capture all the details and nuances of a real world problem in software. However, it is possible to study the problem closely, identify its important points or components, and then create new software data types that represent the essential features or elements of these components. The process of selecting the essential elements of a problem with an eye towards modeling them in software is referred to as *problem abstraction*.

This chapter shows you how to approach the process of problem abstraction. Along the way, you will learn more about classes, methods, fields, Unified Modeling Language (UML) class diagrams, and object-oriented programming. You will learn how to break functionality into logical groups to formulate methods. These methods provide a measure of code reuse that will save you both work and time.

The primary focus of this chapter is the class construct and its use in abstract data type modeling. At the end of the chapter, I present a brief section on structures and explain the similarities and differences between structures and classes. I then offer suggestions on when you might want to implement an abstract data type as a structure and the ramifications of making such a decision.

The material discussed here builds upon that presented in previous chapters. By now you should be very comfortable using your chosen development environment and a handful of .NET Framework classes. You should be able to create simple C# programs, control the flow of program execution with `if`, `if/else`, `for`, `while`, and `do/while` statements, and you should understand the concepts and use of single-dimensional arrays. You should also be an expert at looking through the .NET API documentation for classes that can help you solve problems.

Upon completion of this chapter, you will have added several powerful tools to your programmer's toolbox. These tools will enable you to write increasingly complex programs with ease.

ABSTRACTION: AMPLIFY THE ESSENTIAL, ELIMINATE THE IRRELEVANT

The process of problem abstraction is summarized nicely in the following mantra: *amplify the essential, eliminate the irrelevant*. The very nature of programming demands that a measure of simplification be performed on real world problems. Consider for a moment the concept of numbers. Real numbers can have infinite precision. This means that in the real world, numbers can have an infinite number of digits to the right of the decimal point. This is not possible in a computer with finite resources, therefore the machine representation of real numbers is only an approximation. However, for all practical purposes, an approximation is all the precision required to yield acceptable calculations. In C#, real number approximations are provided by the `float` and `double` data types.

ABSTRACTION IS THE ART OF PROGRAMMING

When compared with all other aspects of programming, problem abstraction requires the most creativity. You must analyze the problem at hand, extract its essential elements, and model these in software. Also, the process of identifying which abstractions to model may entail the creation of software entities that have no corresponding counterpart in the problem domain. Have you ever heard the term, "think outside the box"? It means that to make progress you must shed your old ways of thinking. You must check your prejudices and preconceived notions at the door. Successful programmers have mastered the art of thinking outside, inside, over, under, to the left of, and to the right of the box. With their minds, they transform real world problems into a series of program instructions that are then executed on a machine. Successful programmers have mastered the art of reducing real world problems to a state that can be put inside of a box!

Like any form of art, the mastery of problem abstraction requires lots of practice. The only way to get lots of practice with problem abstraction is to solve lots of problems and write lots of code.

WHERE PROBLEM ABSTRACTION FITS INTO THE DEVELOPMENT CYCLE

Problem abstraction straddles the analysis and design phases of the development cycle. Project requirements may or may not be fully or adequately documented. In fact, on most projects, the important requirements that deeply affect the quality of the source code are not documented at all, and must be derived or deduced from existing or known requirements. Nonetheless, you must be able to distinguish the “signal” of the problem from its “noise”. The abstractions you choose to help model the problem in software directly influence its design (architecture).

CREATING YOUR OWN DATA TYPES

The end result of problem abstraction is the identification and creation of one or more new data types. These data types will interact with each other in some way to implement the solution to the problem at hand. In C#, you create a new data type by defining a new enumeration, structure, class, or interface. Arrays and delegates are data types as well, but not useful for the purposes discussed here. These data types can then be used by other data types. This is referred to as design by composition. The new data types created through the process of problem abstraction are referred to as *abstract data types* or *user-defined types*.

To introduce you to the process of problem abstraction and the creation of new data types, I will walk you through a small case-study project. The rest of this chapter is devoted to developing the data types identified in the project specification along with a detailed discussion about the inner workings of the C# class construct. Most everything you learn about classes also applies to structures. I will discuss the differences between structures and classes at the end of the chapter.

CASE-STUDY PROJECT: WRITE A PEOPLE MANAGER PROGRAM

Figure 9-1 gives the project specification that will be used to build the program presented in this chapter.

People Manager Program

Objectives:

- Apply problem abstraction to determine essential program elements.
- Create user-defined data types using the class construct
- Utilize user-defined data types in a C# application
- Create and manipulate arrays of user-defined data type objects

Tasks:

- Write a simple program that lets you manage people. The program should let users add or delete a person when necessary. The program should also let users set and query a person's last, middle, and first names as well as his or her birthdate and gender. It should also let you determine a person's age.
- Store the people objects in a single-dimensional array.
- Create a separate application class that utilizes the services of a `PeopleManager` class.

Figure 9-1: People Management Program Project Specification

The project specification offers some guidance and several hints. Let's concentrate on the tasks. First, it says that you must write a program to manage people. A full-blown people management program is obviously out of the question, so our first simplification will be to put a bound on exactly what functionality is provided in the final solution. Luckily, we are guided in this decision by the next sentence that says the program should focus on the following functions:

- Add a person
- Delete a person
- Set a person's first, middle, and last names
- Query a person's first, middle, and last names
- Set a person's birthdate
- Query a person's birthdate
- Query a person's gender
- Set a person's gender
- Query a person's age

The project specification also says that you must store person objects in a single-dimensional array. This is clear enough, but where will this array reside? Again, the next sentence provides a clue. It says that you must write a separate application that utilizes the services of a `PeopleManager` class. This is a great hint that provides you with a candidate name for one of the classes that makes up the completed program.

This will suffice for a first-pass analysis of the project specification. The trick now is to derive additional requirements that are not specifically addressed. You can begin by making some assumptions. I recommend you start by identifying the number of classes you will need to write the program. One class, `PeopleManager`, is spelled out for you in the specification. Another class is also alluded to in the last sentence, and that is the application class. You could name the class anything you want, but I will use the name `PeopleManagerApplication`. That should make the purpose of that class clear to anyone reading your code.

OK, you have two classes so far: `PeopleManager` and `PeopleManagerApplication`. Since you will need person objects to work with, you need to create another user-defined type named `Person`. The `Person` class will implement the functionality of a person as required to fulfill the project requirements. You can add additional functionality to exceed the project specification if you desire.

I recommend now that you make a list of the classes identified thus far and assign to them the functionality each requires. One possible list for this project is given in Table 9-1.

Class Name	Functionality Required
Person	<p>The <code>Person</code> class will embody the concept of a person entity. A person will have the following attributes:</p> <ul style="list-style-type: none"> • first name • middle name • last name • gender • birth date <p>The <code>Person</code> class will provide the capability to set and query each of its attributes as well as calculate the age of a person given a person's birth date and the current date.</p>
PeopleManager	<p>The <code>PeopleManager</code> class will manage an array of <code>Person</code> objects. It will have the following attribute:</p> <ul style="list-style-type: none"> • an array of <code>Person</code> objects <p>The <code>PeopleManager</code> class will also provide the following functionality:</p> <ul style="list-style-type: none"> • add a person to the array • delete a person from the array • list the people in the array
PeopleManagerApplication	<p>The <code>PeopleManagerApplication</code> class will be the C# application class that has the <code>Main()</code> method. This class will be used to test the functionality of the <code>PeopleManager</code> and <code>Person</code> classes as they are developed.</p>

Table 9-1: People Manager Program Class Responsibilities

This looks like a good start. As you progress with the design and implementation of each class, especially the `Person` and `PeopleManager` classes, you may find they require functionality not originally thought of or imagined. That's OK — software design is an iterative process. As you progress with the design and implementation of a pro-

gram, you gain a deeper insight or understanding of the problem you are trying to solve. This knowledge is then used to improve later versions of the software. Alright, enough soap boxing! On with the project.

The next step I recommend taking is to examine each class and see which piece of its functionality might be provided by a class from the .NET Framework API. Let's look closely at the Person class. The requirement to calculate a person's age means that we will have to perform some sort of date calculation. The question is, "Is this sort of thing already done for us by the .NET Framework API?" The answer is yes. The place to look for this sort of utility functionality is in the System namespace. There you will find the DateTime class. Take time now to familiarize yourself with the DateTime class, as you will find it helpful in other projects as well.

This completes the analysis phase of this project. You should have a fairly clear understanding of the project requirements and the number of user-defined data types required to implement the solution. The next step I recommend you take is to concentrate on the Person class and implement and test its functionality in its entirety. The Person class is the logical place to start since all the other classes depend on it.

Quick Review

Problem abstraction requires lots of programmer creativity and represents the *art* in the *art of programming*. Your guiding mantra during problem abstraction is to *amplify the essential, eliminate the irrelevant*. Problem abstraction is performed in the analysis and design phase of the development cycle. The abstractions you choose to model a particular problem will directly influence a program's design.

The end result of problem abstraction is the identification and creation of one or more new data types. The data types derived through problem abstraction are referred to as abstract data types (ADTs) or user-defined data types. User-defined data types can be implemented as structures or classes. These structures or classes will interact with each other in some capacity to implement the complete problem solution.

The UML Class Diagram

Now that the three classes of the People Manager project have been identified, you can express their relationship to each other via a UML class diagram. The purpose of a UML class diagram is to express the static relationship between classes, interfaces, and other components of a software system. UML class diagrams are used to communicate and solidify your understanding of software designs to yourself, to other programmers, to management, and to clients. Figure 9-2 gives a basic UML diagram showing the static relationship between the classes identified in the People Manager project.

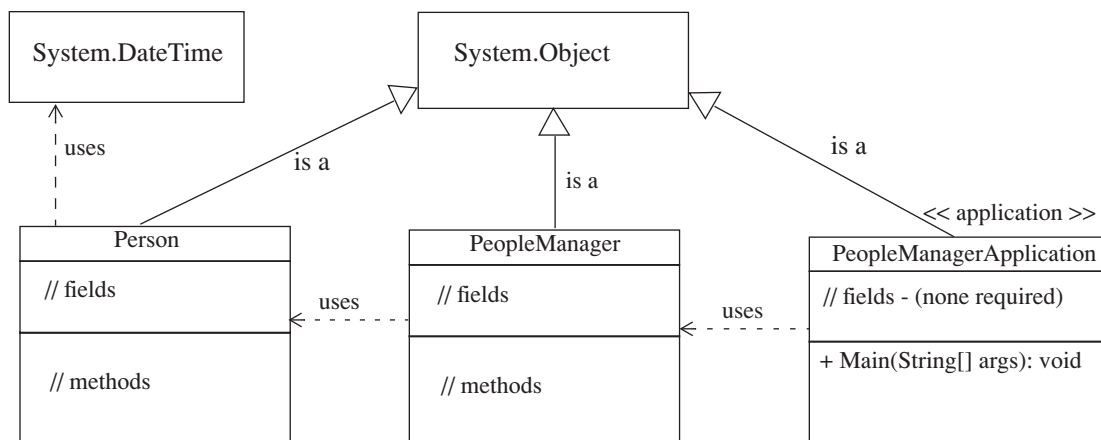


Figure 9-2: Class Diagram for People Manager Classes

Each rectangle shown in Figure 9-2 represents a class. The lines tipped with the hollow arrowheads represent generalization and specialization. The arrow points from the specialized class to the generalized class. This represents an “is a...” relationship between the classes. As Figure 9-2 illustrates, the classes `Person`, `PeopleManager`, and `PeopleManagerApplication` extend the functionality provided by the `System.Object` class. The `Object` class serves as the *direct base class* for all reference types that do not explicitly extend another class. I discuss inheritance in detail in Chapter 11.

Each class rectangle can be drawn either as a simple rectangle or with three compartments. The uppermost compartment will have the class name, the middle compartment will list the fields, and the bottom compartment will list the methods.

Figure 9-2 further shows that the `PeopleManagerApplication` class is an application. This is indicated with the use of the `<<application>>` *stereotype*. A stereotype introduces a new type of element within a system. The name of the new element is contained within the guillemet characters `<< >>`. The application will have one method, `Main()`. Since it is a class, it could have fields and other methods, but in this example no other fields or methods are required.

The `PeopleManagerApplication` class uses the services of the `PeopleManager` class. This is indicated by the dashed arrow pointing from the `PeopleManagerApplication` class to the `PeopleManager` class. The dashed arrow represents a *dependency*. The `PeopleManager` class will have several attributes and methods which have yet to be defined.

The `PeopleManager` class will use the services of the `Person` class. The `Person` class will have fields, properties, and methods as well. These will be developed in the next several sections.

The `Person` class uses the services of the `System.DateTime` structure. The `DateTime` structure will give the `Person` class the ability to calculate the age of each `Person` object.

Now that you have a basic design for the `People Manager` project, you can concentrate on one piece of the design and implement its functionality. Over the next several sections, I discuss the class construct in detail and show you how to create the `Person` and `PeopleManager` classes. Along the way I will show you how to test these classes using the `PeopleManagerApplication` class.

Quick Review

A UML class diagram shows the static relationship between classes that participate in a software design. Programmers use the class diagram to express and clarify design concepts to themselves, to other programmers, to management, and to clients.

In UML, a rectangle represents a class. The rectangle can have three compartments. The uppermost compartment contains the class name, the middle compartment contains fields, and the bottom compartment contains the methods.

A stereotype introduces a new type of element within a system. The stereotype name is contained within the guillemet characters `<< >>`.

Generalization and specialization are indicated by lines tipped with hollow arrows. The arrow points from the specialized class to the generalized class. The generalized class is the base class, and the specialized class is the derived or subclass. Generalizations specify “is a...” relationships between base and subclasses.

Dependencies are indicated by dashed arrows pointing to the class being depended upon. Dependencies are one way to indicate “uses...” relationships between classes.

OVERVIEW OF THE CLASS CONSTRUCT

This section presents an overview of the C# class construct. You have already been exposed to the structure of a C# application class in Chapter 6 so some of this material will be a review.

ELEVEN CATEGORIES OF CLASS MEMBERS

C# classes can contain eleven different types of members: *fields*, *constants*, *methods*, *properties*, *events*, *indexers*, *operators*, *instance constructors*, *static constructors*, *finalizers*, and *nested type declarations*. In this section I

present a brief description of each member type. The rest of the chapter will demonstrate the use of fields, constants, methods, properties, and instance constructors, as these are the most often used class members. I will discuss the remaining class member types later in the book when their use becomes appropriate. I find it best not to present too much information at one go, or your head will explode!

Fields

Fields are variables that are used to set and maintain object state information. Fields can be either *static* or *non-static*.

STATIC OR CLASS-WIDE FIELDS

A static field represents an attribute that is shared among all object instances of a particular class. This means that the field's value exists independently of any particular instance, and therefore does not require a reference to an object to access it. Another term used to describe static fields is *class* or *class-wide fields*.

NON-STATIC OR INSTANCE FIELDS

Non-static fields represent attributes for which each object has its very own copy. Another term used to describe non-static fields is *instance fields*. It's through the use of instance fields that objects can set and maintain their attribute state information. For example, if we are talking about Person objects, each Person object will have its own first name, middle name, last name, gender, and birth date. This instance attribute state information is not shared with other Person objects. Figure 9-3 graphically illustrates the relationship between static and non-static fields.

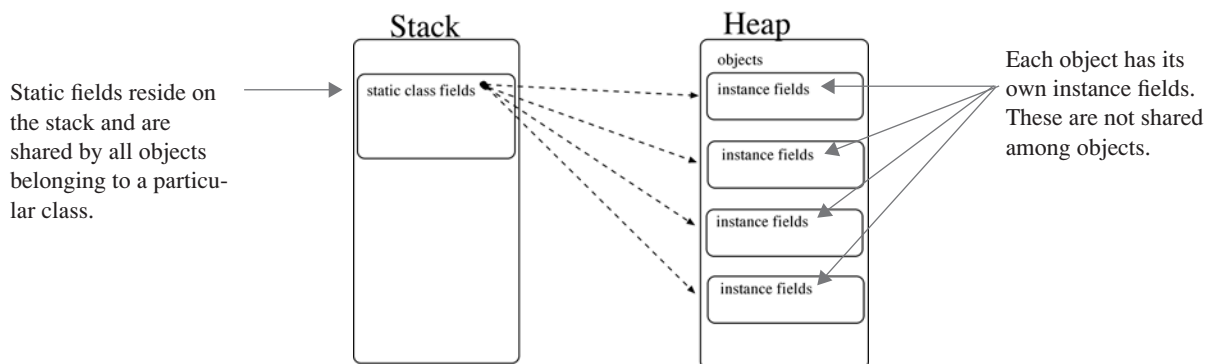


Figure 9-3: Static and Non-Static Fields

READONLY FIELDS

It's often helpful to have a field maintain its first-assigned value throughout the life of the program. Such a field is said to be a *constant*. Instance readonly fields can be initialized at the point of declaration or in one or more constructors, where each constructor might assign a different value to the readonly field. Static readonly fields can be initialized at the point of declaration or in a static constructor. This can be done by declaring a field to be "readonly" with the `readonly` keyword. Let's take a look at the behavior of an ordinary field vs. a readonly field. Example 9.1 offers a simple code example.

9.1 *ReadOnlyTest.cs*

```

1  using System;
2
3  public class ReadOnlyTest {
4      int field_1 = 1;
5      readonly int field_2 = 25;
6
7      static void Main(){
8          ReadOnlyTest rot = new ReadOnlyTest();
9          Console.WriteLine(rot.field_1);
10         Console.WriteLine(rot.field_2);
11     }
12 }
```

Referring to Example 9.1 — two fields have been declared and initialized on lines 4 and 5. The field named `field_2` has been declared `readonly`. This short program simply prints the field values to the console, as is shown in Figure 9-4.

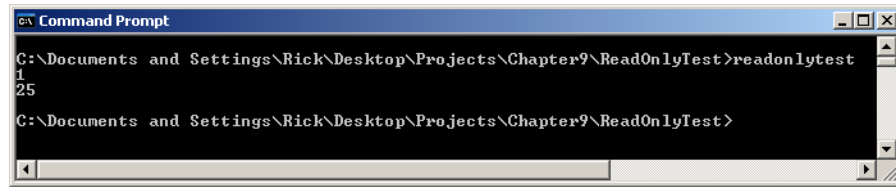


Figure 9-4: Results of Running Example 9.1

As long as you don't try to change the value of a `readonly` field, you'll be fine. Example 9.2 gives a short program that attempts to change the values of both fields. The error produced when I attempt to compile the program is shown in Figure 9-5.

9.2 *ReadOnlyTest.cs (Mod 1)*

```
1  using System;
2
3  public class ReadOnlyTest {
4      int field_1 = 1;
5      readonly int field_2 = 25;
6
7      static void Main(){
8
9          ReadOnlyTest rot = new ReadOnlyTest();
10         Console.WriteLine(rot.field_1);
11         Console.WriteLine(rot.field_2);
12
13         rot.field_1 = 2;
14         rot.field_2 = 26; // this will cause an error
15
16         Console.WriteLine(rot.field_1);
17         Console.WriteLine(rot.field_2);
18     }
19 }
```

Referring to Example 9.2 — the value of `field_1` is modified on line 13 with no problem. The attempt to modify the value of `field_2` results in a compiler error.

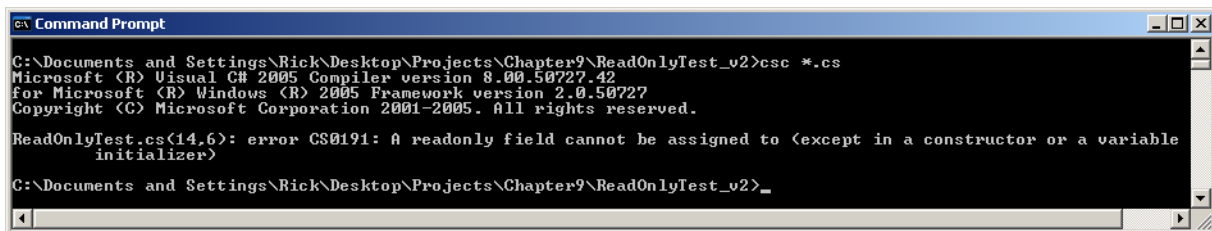


Figure 9-5: Error Resulting from an Attempt to Assign to a `ReadOnly` Field

As you can see from the two previous example programs, a `readonly` field is a constant. Since neither `field_1` nor `field_2` are declared `static`, they are both instance fields, which means that each object of type `ReadOnlyTest` contains its very own copy of these values. It is generally desirable to conserve storage space and declare class constants to be `static`. This way, the constant values are shared among all objects of a particular class, as Figure 9-3 illustrated. To make `field_2` a `static` field, simply add the keyword `static` to the declaration like so:

```
static readonly int field_2 = 25;
```

Doing this, however, changes the field's behavior. Since it's a `static` field, it can only be accessed either via the class name or directly from within a class's `static` or `non-static` methods, as Example 9.3 illustrates.

9.3 *ReadOnlyTest.cs (Mod 2)*

```
1  using System;
2
3  public class ReadOnlyTest {
4      int field_1 = 1;
5      static readonly int field_2 = 25; // now it's a class-wide constant
6
7      static void Main(){
8          ReadOnlyTest rot = new ReadOnlyTest();
9          Console.WriteLine(rot.field_1);
```



```

10     Console.WriteLine(ReadOnlyTest.field_2); // access via classname
11     Console.WriteLine(field_2);             // or directly because it is static!
12 }
13 }

```

Referring to Example 9.3 — adding the `static` keyword to the declaration of `field_2` makes it a static field. Static fields can be accessed directly by a class's static or non-static methods or via the class name, as is shown on line 10. Figure 9-6 gives the results of running this program.

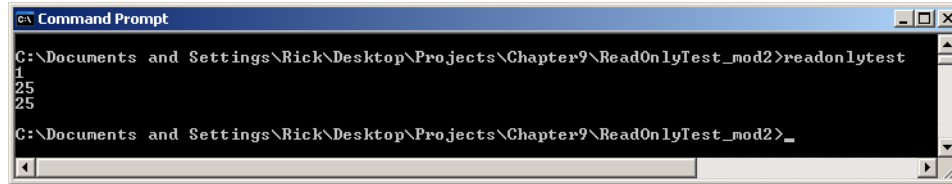


Figure 9-6: Results of Running Example 9.3

CONSTANTS

As you saw in the previous section, class constants can be created by declaring a static readonly field. C# provides a shortcut way to do this with the `const` keyword. Example 9.4 shows how it's done.

9.4 ConstantTest.cs

```

1  using System;
2
3  public class ConstantTest {
4      int field_1 = 1;
5      static readonly int field_2 = 25;
6      const int CONSTANT_1 = 35;
7
8      static void Main(){
9          ConstantTest ct = new ConstantTest();
10         Console.WriteLine(ct.field_1); // can only be accessed via reference since it's non-static
11         Console.WriteLine(ConstantTest.field_2); // can be accessed via class
12         Console.WriteLine(field_2); // or directly because it's static
13         Console.WriteLine(ConstantTest.CONSTANT_1); // can be accessed via class
14         Console.WriteLine(CONSTANT_1); // or directly because it's static
15     }
16 }

```

Referring to Example 9.4 — on line 6 the keyword `const` declares a class-wide constant member. This is akin to declaring a field to be `static readonly`, but there is a difference, which I explain in the next section. Note that uppercase letters were used to form the constant's identifier to make it stand out in the program. Figure 9-7 gives the results of running this program.

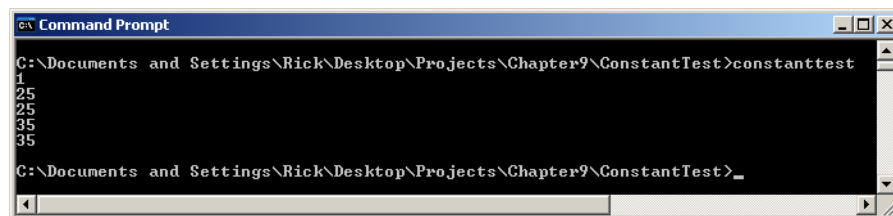


Figure 9-7: Results of Running Example 9,4

THE DIFFERENCE BETWEEN `const` AND `readonly`; *Compile-Time vs. Runtime Constants*

A constant declared with the `const` keyword must be initialized at the moment of declaration. The `const` keyword is used to introduce what are called *compile-time* constants. Use the `readonly` keyword to declare a constant if you need to create the constant object using the `new` keyword or if you need to initialize the constant value in a constructor. For example, if you need to create a constant `DateTime` object that is initialized to a particular date, do something like the following:

```
static readonly DateTime MIN_VALID_SQL_DATE = new DateTime(01, 01, 1753);
```

PROPERTIES

A property is a class member that provides access to an object or class attribute. A property provides accessors that contain statements that are executed when its value is read or written. Properties can be static or non-static, read-only, write-only, or read-write.

Properties, at first glance, can be a confusing concept to grasp. One has a tendency to associate properties with fields, but they are more closely related to methods; property accessors get converted into methods during the compilation process.

INSTANCE PROPERTIES

An instance property is a non-static member that must be accessed via an object reference.

STATIC PROPERTIES

A static property is a class-wide member that can be accessed via the class name or directly in static and non-static methods.

READ-ONLY PROPERTIES

A read-only property is one whose value can only be read and not written. A read-only property defines a `get` accessor.

WRITE-ONLY PROPERTIES

A write-only property is one whose value can only be written and not read. A write-only property defines a `set` accessor.

READ-WRITE PROPERTIES

A read-write property is one whose value can be both read and written. A read-write property defines both a `get` and a `set` accessor.

PROPERTIES IN ACTION

Example 9.5 gives a short program demonstrating the use of properties.

9.5 PropertiesDemo.cs

```

1  using System;
2
3  public class PropertiesDemo {
4
5      /**** Constants and Fields *****/
6      private const String MESSAGE = "Hello Stranger";
7      private static int field_1 = 1;
8      private int field_2 = 2;
9
10     /***** Properties *****/
11     public String ClassName {
12         get { return this.GetType().ToString(); }
13     }
14
15     public String Message {
16         get { return MESSAGE; }
17     }
18
19     public static int ObjectCount {
20         get { return field_1; }
21         set { field_1 = value; }
22     }
23
24     public int SomeProperty {
25         get { return field_2; }
26         set { field_2 = value; }
27     }

```

```

28
29     static void Main(){
30         PropertiesDemo pd = new PropertiesDemo();
31         Console.WriteLine(pd.ClassName);
32         Console.WriteLine(pd.Message);
33         Console.WriteLine(ObjectCount);
34         ObjectCount++;
35         Console.WriteLine(ObjectCount);
36         Console.WriteLine(pd.SomeProperty++);
37         Console.WriteLine(pd.SomeProperty);
38     }
39 }

```

Referring to Example 9.5 — the `PropertiesDemo` class has one constant and two fields. One of the fields, `field_1`, is a static field. I have defined four properties. Note that each property has a type and a name. Property names are by convention formed with *camel case*. Camel case means the first letter of each word in the identifier name is uppercase and the remaining letters of each word are lowercase.

Each property's accessor definitions are enclosed in the property's body, which is denoted by the opening and closing brace. A read-only property has a `get` accessor defined, which itself has an opening and closing brace and can contain any number of statements as long as it eventually returns an object of the property's specified type. For example, the `ClassName` property whose definition begins on line 11 is a read-only property. It defines a `get` accessor that returns a string value. Note that the `ClassName` property computes the value of the string, in this case the class name, by making a series of method calls on the appropriate objects. Compare the behavior of the `ClassName` property to that of the `Message` property whose definition starts on line 15. The `Message` property is a read-only property that simply returns the value of the `MESSAGE` constant.

The `ObjectCount` property is a read-write property because it defines both `get` and `set` accessors. It is also a static property because its definition includes the use of the `static` keyword. Note on line 21 the use of the implicit parameter named "value" in the `set` accessor. Remember that these accessors will be ultimately invoked as method calls. The value parameter is automatically supplied by the compiler when a `set` accessor is called.

The `SomeProperty` property is a read-write instance property.

These four properties are used in the body of the `Main()` method that starts on line 29. Note specifically how properties can be used in ways similar to fields. Instance properties must be accessed in a static method via an object reference. Static properties can be accessed via the class name or directly in static and instance methods. Figure 9-8 gives the results of running Example 9.5.

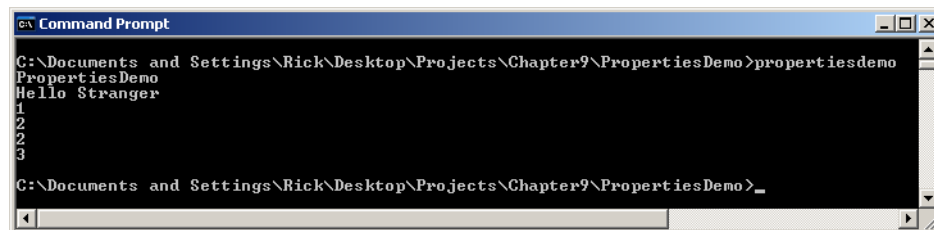


Figure 9-8: Results of Running Example 9.5

Methods

A method is a class member that implements a series of instructions that can be executed or *called* via a class or object. Methods, like fields and properties, can be static or non-static.

Methods can share the same name as long as their method signatures differ. This is referred to as *method overloading*. A method's signature includes its name, and the number and type of its formal parameters. I cover methods in greater detail in this chapter in the Methods section.

INSTANCE CONSTRUCTORS

An *instance constructor*, or simply a *constructor*, is a special type of method that contains the instructions required to properly initialize an object. A constructor method takes the same exact name as the class in which it appears and has no return type.

A *default constructor* is a constructor that has an empty parameter list. (*i.e.*, It takes no parameters.) If you fail to define a default constructor, the compiler will generate one for you.

Constructors, like ordinary methods, can be overloaded. This comes in handy when you want to define several different ways to create an object.

Constructors are usually declared to have public accessibility, although in some cases it's helpful to declare them to be protected or private so you can maintain full control over how and when an instance object is created. (Refer to the Singleton pattern in Chapter 25 for an example.)

Use instance constructors to initialize non-static readonly fields. This is especially helpful if you needed the readonly constant value to be initialized differently according to which constructor was called.

You'll see many examples of instance constructors throughout this book.

STATIC CONSTRUCTORS

A static constructor is a special method that contains the instructions required to properly initialize static class fields. Static constructors take no parameters and are called automatically by the runtime environment when the program executes. The use of access modifiers is not allowed with static constructors. (*i.e.*, A static constructor cannot be public or private.) Use a static constructor if you need to initialize static readonly constants.

EVENTS

An event is a class member that enables a class or an object to provide notifications. I cover events in detail in *Chapter 12 — Windows Forms Programming*, and *Chapter 13 — Custom Events*.

OPERATORS

An operator is a member that defines what it means to apply certain expression operators (*like the '+' or '==' operators for example*) to objects. This *operator overloading* allows you to create well-behaved objects. Operator overloading is covered in detail in *Chapter 21 — Operator Overloading*.

INDEXERS

An indexer is a class member that allows an object to be indexed like an array. I give an example of an indexer in *Chapter 14- Collections*.

NESTED TYPE DECLARATIONS

A type definition that appears within the body of a class is referred to as a *nested type*. The most often seen example of this is when an enumerated type (*enumeration*) is declared within the body of a class. The overuse of nested types leads to hard-to-read and hard-to-maintain code. Generally avoid using them unless you have a compelling reason to do so.

FINALIZERS

A *finalizer* is a class member that's called automatically when an object is collected by the runtime environment garbage collector. A finalizer contains the instructions required to clean up the object. An example of object clean-up might be the release of network resources or file handles used during the object's lifetime.

A finalizer method takes the same name of the class with the tilde character '~' prepended to its name. Finalizers take no parameters and cannot be called explicitly. Because a finalizer method cannot be called explicitly, there is no telling when it will be called. Therefore, the release of critical resources should not be left, as a rule, to the whims of the garbage collector. In practice, use ordinary methods that can be called explicitly to provide critical object clean-up services. The finalizer can then be relied upon as a back-up.

ACCESS MODIFIERS

Use the access modifiers `public`, `protected`, `private`, `internal`, and `protected internal` to control access to class members. If no access is specified, then `private` is assumed. The following sections describe the use of these access modifiers in greater detail.

Public

The keyword `public` indicates that the member is accessible to all client code. Generally speaking, most of the methods, constants, and properties you declare in a class will be `public`.

Private

The keyword `private` indicates that the member is intended for internal class use only and is not available for use by client programs. You will usually declare non-static instance fields to be `private`. You can think of `private` fields as being surrounded by the protective cocoon of the class, though if you're not careful, you can breach this *encapsulation* by absentmindedly returning a reference to a `private` field via a method or property.

You can also declare methods to be `private` as well. `Private` methods are intended to be utilized exclusively by other methods within the class. These `private` methods are often referred to as *utility methods* since they are usually written to perform some utility function that is not intended to be part of the class's public interface.

Protected

The keyword `protected` prevents horizontal access to members, but allows them to be inherited or accessed by subclasses. I discuss the `protected` keyword in detail in *Chapter 11 — Inheritance*.

Internal

The meaning of the `internal` keyword is “for use within this program”. An alternative meaning for the `internal` keyword might be “local public”. Essentially, if you declare a member to have `internal` accessibility, it can be freely accessed by other classes and members within the same assembly. This includes separate `.netmodules` that are compiled together using the `/addmodule` compiler switch.

If, however, you create a dynamic link library (dll) using the `/target:library` compiler switch, then `internal` members are accessible to other `internal` classes and members within that dll, but are not available for use by external code that links to it.

Protected Internal

A member declared to have `protected internal` accessibility is visible to all components contained within the dll just as the `internal` keyword specifies. Additionally, it can be inherited by subclasses of the member's containing type regardless of the assembly to which the subclass belongs.

THE CONCEPTS OF HORIZONTAL ACCESS, INTERFACE, AND ENCAPSULATION

The term *horizontal access* describes the access a client object has to the members of a server object. The client object represents the code that uses the services of another object. It can do this in two ways: 1) by accessing a class's public static members via the class name, or 2) by creating an instance of the class and accessing its public non-static members via an object reference.

The members (usually constants, properties, methods, constructors, and events) a class exposes as `public` are collectively referred to as its *public interface*. Client code becomes dependent upon these public interface members. The wrong kind of change to a class's interface will break any code that depends upon that interface. When changing a class's public interface, the rule-of-thumb is that you can add public members but never remove them. If you look through the .NET API you will see lots of classes with *deprecated* members. A *deprecated* member is a member that

is targeted for deletion in some future version of the API. These members are not yet removed because doing so would break existing programs that use (depend upon) those members.

Any member declared `private` is said to be encapsulated within its class, as it is shielded from horizontal access by client code. Generally speaking, a class's interface can be thought of as the set of services it provides to client programs. It provides those services by manipulating its private, or encapsulated, data structures. The idea is that at some point in the future, programmers may think up a new ways to enhance a particular service's functionality. They may do this by making changes to the class's internal, or private, data structures. Since these data structures are encapsulated, a change to them will have no effect on client code, except perhaps for the effects of an improvement to the service provided.

Figure 9-9 illustrates the concept of horizontal access and the effects of using `public` and `private` access modifiers.

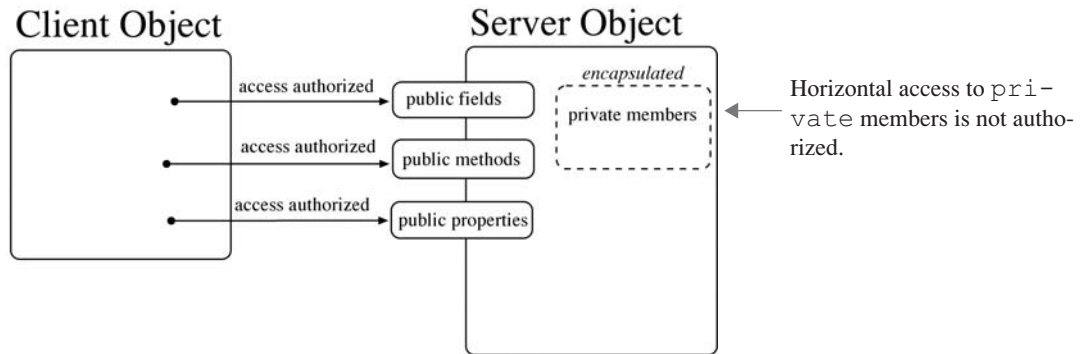


Figure 9-9: Horizontal Access Controlled via Access Modifiers `public` and `private`

The concepts of public interfaces, horizontal access, and encapsulation are important to the world of object-oriented programming, which means they are important to you. You will deal with these concepts every time you write code in the C# language.

Quick Review

C# classes can contain eleven different types of members: *fields*, *constants*, *methods*, *properties*, *events*, *indexers*, *operators*, *instance constructors*, *static constructors*, *finalizers*, and *nested type declarations*.

The access modifiers `public`, `protected`, `private`, `internal`, and `protected internal` are used to control access to class and instance members. If no access is specified then `private` is assumed.

The term *horizontal access* describes the access a client object has to the members of a server object. The client object represents the code that uses the services of another object. It can do this in two ways: 1) by accessing a class's public static members via the class name, or 2) by creating an instance of the class and accessing its public non-static members via an object reference.

Methods

A method is a named module of executable program functionality. A method contains program statements that, when grouped together, represent a basic level of code reuse. Access the functionality of a method by calling the method using its name in a program. I use the term program here to mean any piece of code that could possibly use the services of the class that defines the method. This might include 1) another method within the class you are defining, 2) another class within your program, or 3) a third-party program that wants to use the services provided by your program.

In the C# language, a method must belong to a class; methods cannot exist or be defined outside of a class construct.

METHOD NAMING: USE ACTION WORDS THAT INDICATE THE METHOD'S PURPOSE

Use action words (verbs) when naming a method that provide an indication of the method's intended purpose. See *Appendix C: Identifier Naming and Self-Commenting Code* for a detailed discussion on how to formulate identifier names in a way that makes your code humanly readable.

MAXIMIZE METHOD COHESION

The first rule of thumb to keep in mind when writing a method is to keep the functionality of the method focused on the task at hand. The formal term used to describe a method's focus characteristic is *cohesion*. Your goal is to write highly cohesive methods. A method that does things it really shouldn't be doing is not focused and is referred to as *minimally cohesive*. You can easily write cohesive methods if you follow this two-step approach:

- Step 1: Follow the advice offered in the previous subsection and start with a good method name. The name of the method must indicate the method's intended purpose.
- Step 2: Keep the method's body code focused on performing the task indicated by the method's name. A well-named, maximally-cohesive method pulls no surprises!

Sounds simple enough. But if you're not careful, you can slip functionality into a method that doesn't belong there. Sometimes you will do this because you are lazy, and sometimes it will happen no matter how hard you try to avoid doing so. Practice makes perfect!

STRUCTURE OF A METHOD DEFINITION

A method definition declares and implements a method. A method definition consists of several optional and mandatory components. These include method modifiers, a return type or void, method name, and parameter list. I discuss these method components in detail below. Figure 9-10 shows the structure of a method definition.

```
method_modifiersopt return_type or voidopt method_name( parameter_listopt ){
    // method body - program statements go here
}
```

Figure 9-10: Method Definition Structure

Any piece of the method definition structure shown in Figure 9-10 that's labeled with the subscript *opt* is optional and can be omitted from a method definition depending on the method's required behavior. In this chapter, I focus on just a few of the potentially many method variations you can write. You will be gradually introduced to different method variations as you progress through the book. The following sections describe each piece of the method definition structure in more detail.

Method Modifiers (optional)

Use method modifiers to specify a particular aspect of method behavior. Table 9-2 lists and describes the C# keywords that can be used as method modifiers.

Modifier	Description
public	The <code>public</code> keyword declares the method's accessibility to be public. Public methods can be accessed by client code (<i>i.e.</i> , grants horizontal access to the method).

Table 9-2: Method Modifiers

Modifier	Description
protected	The <code>protected</code> keyword declares the method's accessibility to be protected. Protected accessibility prevents horizontal access but allows the method to be inherited by derived classes.
private	The <code>private</code> keyword declares the method's accessibility to be private. It prevents both horizontal access and method inheritance.
internal	The meaning of the <code>internal</code> keyword is "for use within this program". Internal methods can be freely accessed by other classes and members within the same assembly. This includes separate .netmodules compiled together using the <code>/addmodule</code> compiler switch. If, however, you create a dynamic link library (dll) using the <code>/target:library</code> compiler switch, then internal methods are accessible to other internal classes and members within that dll, but are not available for use by external code that links to it.
protected internal	A method declared to have <code>protected internal</code> accessibility is visible to all components contained within the assembly as specified by the <code>internal</code> keyword, and for use (<i>i.e.</i> , can be inherited) by subclasses of the member's containing type, regardless to which assembly the subclass belongs.
static	The <code>static</code> keyword declares a static or class method.
abstract	The <code>abstract</code> keyword declares a method that contains no body (no implementation). The purpose of an abstract method is to defer the implementation of a method's functionality to a subclass. Abstract methods are discussed in <i>Chapter 11 — Inheritance</i> .
new	The <code>new</code> keyword declares a member with the same name or method signature as an inherited member. This new member hides the base member. The <code>new</code> keyword is covered in detail in <i>Chapter 11 — Inheritance</i> .
override	The <code>override</code> keyword designates a method as overriding an inherited method. The <code>override</code> keyword is covered in detail in <i>Chapter 11 — Inheritance</i> .
virtual	The <code>virtual</code> keyword designates a method as being virtual. A virtual method can be overridden in a subclass. The <code>virtual</code> keyword is covered in detail in <i>Chapter 11 — Inheritance</i> .
sealed	The <code>sealed</code> keyword prevents a method from being overridden in derived classes. Sealed methods are covered in detail in <i>Chapter 11 — Inheritance</i> .
extern	The <code>extern</code> keyword designates a method as being external. An external method is one that is implemented in a language other than C#. (C++ or C for example) The use of the <code>extern</code> keyword is not covered in this book.

Table 9-2: Method Modifiers

RETURN TYPE OR Void (optional)

A method can return a result as a side effect of its execution. If you intend for a method to return a result, you must specify the return type of the result. If the method does not return a result, then you must use the keyword `void`.

The return type and `void` are optional because constructor methods return neither. Constructor methods are discussed in detail later in this section.

Method Name (MANDATORY)

The method name is mandatory. As I discussed earlier, you should use verbs in method names since methods perform some sort of action. If you chose to ignore good method naming techniques, you will find that your code is hard, if not impossible, to read and understand. As a result, it will also be hard to fix if it's broken.

Parameter List (optional)

A method can specify one or more formal parameters. Each formal parameter has a type, a name, and an optional modifier (`ref` or `out`). The name of the parameter has local scope within the body of the method and hides any field members having the same name. By default, arguments are passed to method parameters by value. The `ref` parameter modifier can be used to pass arguments by reference. The `out` parameter modifier is used to return values to the calling program via the method arguments.

Method Body (optional for abstract or external methods)

The method body is denoted by a set of opening and closing brackets. Any code that appears between a method's opening and closing brackets is said to be in the body of the method. If you are declaring an abstract or external method, omit the braces and terminate the method declaration with a semicolon.

Method Definition Examples

This section offers a few examples of method definitions. The body code is omitted so that you can focus on the structure of each method definition. The following line of code would define a method that returns a `String` object that represents the first name of some object (perhaps a `Person` object).

```
public String GetFirstName(){ // body code goes here }
```

Notice that the above method definition uses the `public` access modifier, declares a return type of `String`, and takes no arguments because it declares no parameters. The name of the method is `GetFirstName`, which does a good job of describing the method's purpose.

The next method declaration might be used to set an object's first name:

```
public void SetFirstName(String first_name){ // body code goes here }
```

This method is also public, but it does not return a result, hence the use of the keyword `void`. It contains one parameter named `first_name` that is of type `String`.

The following method definition might be used to get a `Person` object's age:

```
public int GetAge(){ // body code goes here }
```

This method is public and returns an integer type result. It takes no arguments.

See if you can guess what type of method is being defined by the following definition:

```
public Person(String f_name, String m_name, String l_name){
    // body code goes here
}
```

If you guessed that it was a constructor method, you would be right. Constructor methods have no return type, not even `void`. This particular constructor declares three formal parameters having type `String`.

METHOD SIGNATURES

Methods have a distinguishing characteristic known as a signature. A method's signature consists of its name and the number, modifiers, and types of its parameters. Method modifiers and return types are not part of a method's signature. It's important that you understand the concept of method signatures so that you can understand the concept of method overloading, which is discussed in the next section.

Methods with different names and the same parameter list have different signatures. Methods with the same name and different parameter lists have different signatures as well, and are said to be overloaded (because they share the same name). Methods cannot have the same name and identical parameter lists. This will cause a compiler error.

OVERLOADING METHODS

A class can define more than one method with the same name but having different signatures. This is referred to as method *overloading*. You would overload methods when the method performs the same function but in a slightly different way or on different argument types. The most commonly overloaded method is the class constructor. You will see many examples of overloaded class constructors throughout the remaining chapters of this book.

Another frequently encountered method overloading scenario occurs when you want to provide a public method for horizontal access but actually do the work behind the scenes with a private method. The only rule, as stated above, is that each method must have a different signature, which means their names can be the same but their parameter lists must be different in some way. The fact that one is public and the other is private has no bearing on their signatures.

CONSTRUCTOR METHODS

Constructor methods are special methods whose purpose is to set up or build the object in memory when it is created. You can choose not to define a constructor method for a particular class if you desire. In that case, the compiler creates a default constructor for you. This default constructor will usually not provide the level of functionality you require except perhaps in the case of very simple or trivial class declarations. If you want to be sure of the state of an object when it is created, you must define one or more constructor methods.

Quick REVIEW

Methods are named modules of executable program functionality. Methods contain program statements that, when grouped together, represent a basic level of code reuse. You access the functionality of a method by calling the method using its name in a program.

Methods should be well named and maximally cohesive. A well named, maximally cohesive method will pull no surprises.

Method definitions have structure. Their behavior can be optionally modified with method modifiers, they can optionally specify a return result type or `void`, and they can have an optional parameter list.

Methods have a distinguishing characteristic known as a method signature. Methods with different names and parameter lists are said to have different signatures. Methods with different names and the same parameter list also have different signatures. Methods with the same name and different parameter lists have different signatures as well and are said to be overloaded (because they share the same name). Methods cannot have the same name and identical parameter lists. This will cause a compiler error.

Constructor methods set up or build an object when it's created in memory. If you do not provide one, the compiler will create a default constructor for you, but it may or may not provide the level of functionality you require.

Building And Testing The Person Class

Now that you have been introduced to the C# class construct in more detail, it's time to apply some of what was discussed above to create and test the Person class. To get the Person class in working order, you will have to take the results of the analysis performed earlier and map attributes and functionality to fields, properties, and methods. As you build the Person class, you may discover that you need to add members as necessary to fully implement the class to your satisfaction. That's a normal part of the design and programming process.

To write the code for the Person class, I will use the development cycle presented in Chapter 1 and explained in detail in Chapter 3. The development cycle is applied iteratively. (*i.e.*, I will apply the steps of plan, code, test, and integrate repeatedly until I have completed the code.)

Start By Creating The Source File And Class Definition Shell

I recommend you start this process by creating the Person.cs source file and the Person class definition shell. Example 9.6 gives the code for this early stage of the program.

9.6 Person.cs (1st Iteration)

```
1  public class Person {  
2  
3  } //end Person class
```

At this point, I recommend you compile the code to ensure you've typed everything correctly and that the name of the class matches the name of the file. Because you are defining a class that contains no Main() method, you'll get an error stating such unless you create a module using the /target:module compiler switch. The complete command required to compile the Person.cs file will be:

```
csc /target:module Person.cs
```

A successful compilation results in no errors or warnings, and a file named Person.netmodule will be written to the project directory.

The next thing to do is to refer to Table 9-1 and see what attributes or fields the Person class must contain, and add those fields. This is done in the next section.

Defining Person Instance Fields

After consulting Table 9-1, you learn that the Person class represents a person entity in our problem domain. Each person has his or her own name, gender, and birth date, so these are good candidates for instance fields in the Person class. Example 9.7 shows the Person class code after the instance fields have been added.

9.7 Person.cs (2nd Iteration)

```
1  using System;  
2  
3  public class Person {  
4      private String  _firstName;  
5      private String  _middleName;  
6      private String  _lastName;  
7      private String  _gender;  
8      private DateTime _birthday;  
9  
10 } // end Person class
```

Two .NET API classes are used for the fields in Example 9.7: String and DateTime. The using keyword on line 1 provides shortcut naming for both the String and DateTime types. These fields represent a first attempt at defining fields for the Person class. Each field is declared to be private, which means they will be encapsulated by the Person class to prevent horizontal access. The only way to access or modify these fields will be through the Person class's public interface properties or methods. Let's define a few of those right now. But, before you move on, compile the Person.cs source file again to make sure you didn't break anything. Compiling the Person class in its present state will result in several compiler warnings, one for each unused field. You may safely ignore those warnings for now.

DEFINING PERSON PROPERTIES AND CONSTRUCTOR METHOD

Now that several Person class instance fields have been created, it's time to define a way to set and manipulate those fields. My approach starts by defining Person's instance properties. I will then use the properties in one or more constructor methods to set the value of each field at the time of object creation. After you've defined several properties and a constructor method, you can use them to test those aspects of Person class behavior.

Adding PROPERTIES

Properties are the preferred way to get or set an object's attributes. In this simple example, the initial set of properties defined for the Person class will correspond to its instance fields. Example 9.8 shows the code for an initial set of read/write properties.

9.8 Person.cs (3rd Iteration)

```

1  using System;
2
3  public class Person {
4
5      // private instance fields
6      private String _firstName;
7      private String _middleName;
8      private String _lastName;
9      private String _gender;
10     private DateTime _birthday;
11
12
13     // public properties
14     public String FirstName {
15         get { return _firstName; }
16         set { _firstName = value; }
17     }
18
19     public String MiddleName {
20         get { return _middleName; }
21         set { _middleName = value; }
22     }
23
24     public String LastName {
25         get { return _lastName; }
26         set { _lastName = value; }
27     }
28
29     public String Gender {
30         get { return _gender; }
31         set { _gender = value; }
32     }
33
34     public DateTime BirthDay {
35         get { return _birthday; }
36         set { _birthday = value; }
37     }
38
39 } // end Person class

```

Referring to Example 9.8 — each read/write property implements a get and set accessor. Remember that the identifier named “value” is an implied parameter. Compiling the Person.cs file in its current state will clear up the unused field warnings now that each field is used in a property definition.

Adding A CONSTRUCTOR METHOD

The purpose of a constructor method is to properly initialize an object when it is created in memory. In the case of the Person class, this means that each person object's fields must be initialized to some valid value. To make this happen, I will add a constructor method that takes a parameter list matching the fields contained in the Person class. These parameters will then be used to initialize each field. The approach I will take will be to initialize the fields via the properties. Example 9.9 gives the code for the Person class definition after the constructor has been added.

9.9 Person.cs (4th Iteration)

```

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

```

Referring to Example 9.9 — the Person constructor method begins on line 13. Notice that it is declared to be public and has no return value. It has five parameters. Each parameter is used in the body of the constructor to set the Person’s properties. The properties, in turn, set the values of their corresponding fields.

OK, now that you’ve got the constructor written, compile the Person.cs source file to ensure you didn’t break anything. It’s now time to test this puppy.

TESTING THE PERSON CLASS: A MINIATURE TEST PLAN

Testing the Person class at this stage of the development cycle consists of creating a Person object and then writing and reading each of its properties. When you create a Person object using the constructor defined in the previous section, you are testing that constructor. The constructor method exercises each property’s set accessor. Printing the value of each property to the console would test each property’s get accessor.

Use The PeopleManagerApplication Class As A Test Driver

To test the Person class functionality you’ll need to create an application class. Since you need to create the PeopleManagerApplication class anyway, you may as well use that class as a test driver. The term *driver* means a small program written specifically to run or test another program. Example 9.10 gives the code for the PeopleManagerApplication class with a few lines of code that tests the functionality of the Person class developed thus far.

9.10 *PeopleManagerApplication.cs*
(Testing *Person*)

```

1  using System;
2
3  public class PeopleManagerApplication {
4      public static void Main(){
5          Person p1 = new Person("Ulysses", "S", "Grant", "Male", new DateTime(1822, 04, 22));
6          Console.WriteLine(p1.FirstName + " " + p1.MiddleName + " " + p1.LastName + " "
7                          + p1.Gender + " " + p1.BirthDay);
8      } // end Main
9  } // end class definition

```

Referring to Example 9.10 — notice how a new `DateTime` object must be created before being used as an argument for the `Person` constructor method. To compile this program with the `Person.netmodule`, use the following command:

```
csc /addmodule:Person.netmodule PeopleManagerApplication.cs
```

Figure 9-11 shows the results of running this program. Everything appears to run fine. It's now time to add a few more features to the `Person` class.

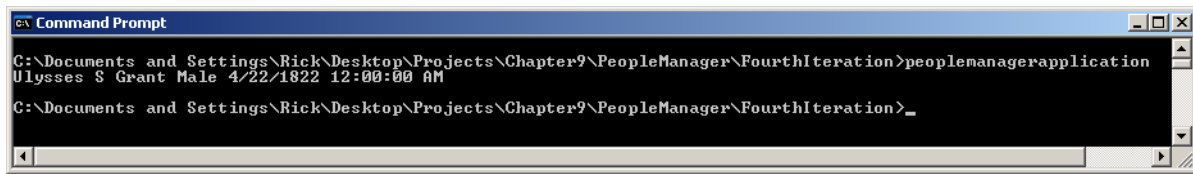


Figure 9-11: Results of Running Example 9.10

Adding FEATURES TO THE PERSON CLASS: CALCULATING AGE

Returning to Table 9-1 for some direction reveals the requirement to calculate a person's age. This could be done in several ways. Think for a moment how you might go about doing this in real life. You might ask people for their birth date and perform the calculation yourself, or you could just ask them how old they are and let them do the calculation for you. I will take the later approach. I'll add a read-only property named `Age` that computes a `Person` object's age and returns the result. Example 9.11 shows the modified `Person` class code.

9.11 *Person.cs* (5th Iteration)

```

1  using System;
2
3  public class Person {
4
5      // private instance fields
6      private String _firstName;
7      private String _middleName;
8      private String _lastName;
9      private String _gender;
10     private DateTime _birthday;
11
12     public Person(String firstName, String middleName, String lastName,
13                 String gender, DateTime birthday){
14         FirstName = firstName;
15         MiddleName = middleName;
16         LastName = lastName;
17         Gender = gender;
18         BirthDay = birthday;
19     }
20
21     // public properties
22     public String FirstName {
23         get { return _firstName; }
24         set { _firstName = value; }
25     }
26
27     public String MiddleName {
28         get { return _middleName; }
29         set { _middleName = value; }
30     }
31
32     public String LastName {
33         get { return _lastName; }
34         set { _lastName = value; }
35     }

```

```

36
37     public String Gender {
38         get { return _gender; }
39         set { _gender = value; }
40     }
41
42     public DateTime BirthDay {
43         get { return _birthday; }
44         set { _birthday = value; }
45     }
46
47     public int Age {
48         get {
49             int years = DateTime.Now.Year - _birthday.Year;
50             int adjustment = 0;
51             if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
52                 adjustment = 1;
53             }
54             return years - adjustment;
55         }
56     }
57 } // end Person class

```

Referring to Example 9.11 — the Age property definition begins on line 47. As you can see, calculating someone's age takes some doing.

After making the necessary modifications to the Person class you can test the changes in the PeopleManagerApplication class. Example 9.12 shows the code for the modified PeopleManagerApplication class. Figure 9-12 shows the results of running this program.

9.12 PeopleManagerApplication.cs
(Testing Person Age property)

```

1  using System;
2
3  public class PeopleManagerApplication {
4      public static void Main(){
5          Person p1 = new Person("Ulysses", "S", "Grant", "Male", new DateTime(1822, 04, 22));
6          Console.WriteLine(p1.FirstName + " " + p1.MiddleName + " " + p1.LastName + " "
7                          + p1.Gender + " " + p1.BirthDay);
8          Console.WriteLine(p1.FirstName + " is " + p1.Age + " years old!");
9      } // end Main
10 } // end class definition

```

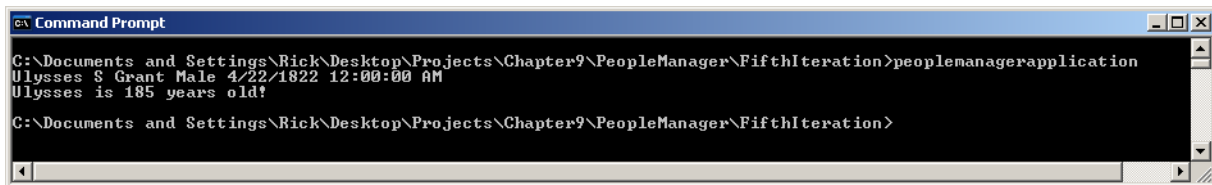


Figure 9-12: Results of Running Example 9.12

Adding FEATURES TO THE PERSON CLASS: CONVENIENCE PROPERTIES

The Age property seems to work pretty well. However, it's a hassle to get a Person object's full name and other vital information by calling each individual property. It might be a good idea to add a property that will do the job for you. While you're at it, you could add a property that returns both the full name and age. Each of these properties can use the services of existing properties. Example 9.13 shows the modified Person class.

9.13 Person.cs (6th Iteration)

```

1  using System;
2
3  public class Person {
4
5      // private instance fields
6      private String _firstName;
7      private String _middleName;
8      private String _lastName;
9      private String _gender;
10     private DateTime _birthday;
11
12     public Person(String firstName, String middleName, String lastName,
13                 String gender, DateTime birthday){
14         FirstName = firstName;

```

```

15     MiddleName = middleName;
16     LastName = lastName;
17     Gender = gender;
18     BirthDay = birthday;
19 }
20
21 // public properties
22 public String FirstName {
23     get { return _firstName; }
24     set { _firstName = value; }
25 }
26
27 public String MiddleName {
28     get { return _middleName; }
29     set { _middleName = value; }
30 }
31
32 public String LastName {
33     get { return _lastName; }
34     set { _lastName = value; }
35 }
36
37 public String Gender {
38     get { return _gender; }
39     set { _gender = value; }
40 }
41
42 public DateTime BirthDay {
43     get { return _birthday; }
44     set { _birthday = value; }
45 }
46
47 public int Age {
48     get {
49         int years = DateTime.Now.Year - _birthday.Year;
50         int adjustment = 0;
51         if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
52             adjustment = 1;
53         }
54         return years - adjustment;
55     }
56 }
57
58 public String FullName {
59     get { return FirstName + " " + MiddleName + " " + LastName; }
60 }
61
62 public String FullNameAndAge {
63     get { return FullName + " " + Age; }
64 }
65
66 } // end Person class

```

Referring to Example 9.13 — the `FullName` property appears on line 58. It concatenates the `FirstName`, `MiddleName`, and `LastName` properties and returns the resulting `String` object that represents the `Person` object's full name.

The `FullNameAndAge` property on line 62 utilizes the services of the `FullName` and `Age` properties. This is a good example of code reuse at the class level. Since the properties exist and already provide the required functionality it's a good idea to use them.

It's time to compile the `Person` class and test the changes. Example 9.14 gives the modified `PeopleManagerApplication` class with the changes required to test the `Person` class's new functionality. Notice the code is a lot cleaner now. Figure 9-13 shows the results of running this program.

9.14 *PeopleManagerApplication.cs*
(Testing *Person FullNameAndAge* Property)

```

1  using System;
2
3  public class PeopleManagerApplication {
4      public static void Main(){
5          Person p1 = new Person("Ulysses", "S", "Grant", "Male", new DateTime(1822, 04, 22));
6          Console.WriteLine(p1.FullNameAndAge);
7      }
8  }

```

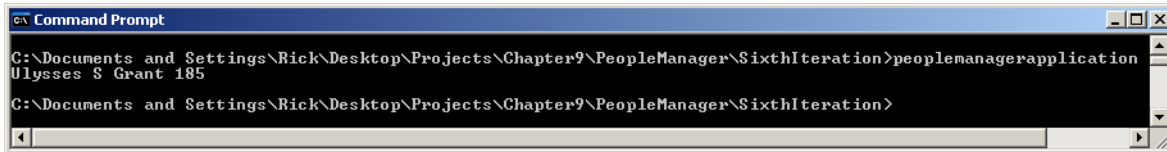


Figure 9-13: Results of Running Example 9.14

Adding FEATURES TO THE PERSON CLASS: FINISHING TOUCHES

It's time to take a step back and look at the Person class with an eye towards adding any methods, properties, or other members that might make its usage easier or more intuitive. There are, in fact, many ways to improve upon the design of Person class, but some of what can be done will have to wait until you've gone a little farther in the book.

One aspect of performance we can address here is the addition of a default constructor. Because up to this point I have failed to implement a default constructor, Person objects can be created with meaningless values assigned to each field. To prevent this from happening, add a private default constructor. This will force the use of the one public constructor currently supported by the Person class.

Another helpful member to add is an overriding ToString() method. Although I do not formally cover the concept of method overriding until *Chapter 11 — Inheritance*, it won't hurt to give you a peek at a simple example.

One last thing. It would be nice to limit the range of authorized values the Gender property can assume. This is a perfect use for an enumeration. Example 9.15 gives the code for the improved Person class.

9.15 Person.cs (7th Iteration)

```

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     //private default constructor
16     private Person(){}
17
18     public Person(String firstName, String middleName, String lastName,
19                   Sex gender, DateTime birthday){
20         FirstName = firstName;
21         MiddleName = middleName;
22         LastName = lastName;
23         Gender = gender;
24         BirthDay = birthday;
25     }
26
27     // public properties
28     public String FirstName {
29         get { return _firstName; }
30         set { _firstName = value; }
31     }
32
33     public String MiddleName {
34         get { return _middleName; }
35         set { _middleName = value; }
36     }
37
38     public String LastName {
39         get { return _lastName; }
40         set { _lastName = value; }
41     }
42
43     public Sex Gender {
44         get { return _gender; }
45         set { _gender = value; }
46     }
47
48     public DateTime BirthDay {

```

```

49     get { return _birthday; }
50     set { _birthday = value; }
51 }
52
53     public int Age {
54         get {
55             int years = DateTime.Now.Year - _birthday.Year;
56             int adjustment = 0;
57             if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
58                 adjustment = 1;
59             }
60             return years - adjustment;
61         }
62     }
63
64     public String FullName {
65         get { return FirstName + " " + MiddleName + " " + LastName; }
66     }
67
68     public String FullNameAndAge {
69         get { return FullName + " " + Age; }
70     }
71
72     public override String ToString(){
73         return FullName + " is a " + Gender + " who is " + Age + " years old.";
74     }
75
76 } // end Person class

```

Referring to Example 9.15 — the enumeration `Sex` is defined on line 6 and provides two authorized values: `MALE` and `FEMALE`. The `_gender` field's type on line 12 is now `Sex` vs. `String`. A similar change was made to the constructor's gender parameter. The private default constructor appears on line 17. On line 43, the type of the `Gender` property was changed to `Sex`. Finally, the overriding `ToString()` method definition begins on line 72.

Example 9.16 shows the modified `Person` class being tested in the `PeopleManagerApplication` class.

*9.16 PeopleManagerApplication.cs
(Testing modified Person class)*

```

1     using System;
2
3     public class PeopleManagerApplication {
4         public static void Main(){
5             Person p1 = new Person("Ulysses", "S", "Grant", Person.Sex.MALE, new DateTime(1822, 04, 22));
6             Console.WriteLine(p1);
7         } // end Main
8     } // end class definition

```

Referring to Example 9.16 — note the use of the enumeration in the constructor argument list to set the `Person` object's gender. Also note now that because `Object`'s `ToString()` method has been overridden, all that's required to print a `Person` object's vital information is to simply call the `WriteLine()` method with the argument `p1`. Figure 9-14 shows the results of running this program.



Figure 9-14: Results of Running Example 9.16

Quick Review

Incrementally build and test abstract data types by iteratively applying the steps of the development cycle. Start with the class definition shell and then add fields, properties, and methods as required to fulfill the class's design objectives.

Test class functionality with the help of a test driver. A test driver is a small program that's used to exercise the functionality of another program.

BUILDING AND TESTING THE PEOPLEMANAGER CLASS

Now that the Person class is finished, it's time to shift focus to the PeopleManager class. Consulting Table 9-1 again reveals that the PeopleManager class will manipulate an array of Person objects. It must insert Person objects into the array, delete Person objects from the array, and list the names and perhaps other information for Person objects contained in the array.

The same approach used to develop the Person class is used here to develop the PeopleManager class. The development cycle is applied iteratively to yield the final result.

DEFINING THE PEOPLEMANAGER CLASS SHELL

Example 9.17 gives the source code for the PeopleManager class definition shell. Compile the code the same way you did the Person class by using the compiler's `/target:module` switch.

9.17 PeopleManager.cs (1st Iteration)

```
1  public class PeopleManager {
2
3
4
5  } // end PeopleManager class
```

To this shell you will add fields and methods.

DEFINING PEOPLEMANAGER FIELDS

Table 9-1 says the PeopleManager class will manage an array of Person objects. This means it will need a field that is a single-dimensional array of type Person. Example 9.18 gives the modified source code for the PeopleManager class after the declaration of a Person array named `people_array`.

9.18 PeopleManager.cs (2nd Iteration)

```
1  public class PeopleManager {
2      Person[] people_array;
3
4
5  } // end PeopleManager class
```

Additional fields may be required, but for now this is a good start. You can compile this file in its current state using two approaches. If you want to use the Person.netmodule created earlier you can use the following compiler command:

```
csc /target:module /addmodule:Person.netmodule PeopleManager.cs
```

Alternatively, you can compile both the Person.cs and PeopleManager.cs files together using the following compiler command:

```
csc /target:module Person.cs PeopleManager.cs
```

Both approaches yield a new module named PeopleManager.netmodule. Now it's time to add some methods.

DEFINING PEOPLEMANAGER CONSTRUCTOR METHODS

We'll give the PeopleManager class two constructors. One will be a default constructor, but unlike the Person's private default constructor, this one will be public for the reasons you'll soon see. The other constructor will do most of the dirty work of initializing the PeopleManager object. This includes initializing the `people_array` and any other fields we add to the PeopleManager class. The default constructor will simply call the other constructor with a default array length value.

To create the `people_array` object, you will need to know how long the array must be. (*i.e.*, how many Person references you need it to store.) You will supply the length via a constructor parameter. If you do not supply a length argument when you create an instance of PeopleManager then the default constructor will create the `people_array`

with some default length value. In the following example, I use a default length of 10. Example 9.19 gives the modified PeopleManager class after the constructors have been added.

9.19 PeopleManager.cs (3rd Iteration)

```

1  using System;
2
3  public class PeopleManager {
4      // private fields
5      private Person[] people_array;
6
7      // overloaded constructor
8      public PeopleManager(int length){
9          people_array = new Person[ length];
10     }
11
12     // default constructor
13     public PeopleManager():this(10){ }
14
15 } // end PeopleManager class

```

Referring to Example 9.19 — the constructor on line 8 takes an integer parameter named length and uses it to dynamically create the people_array in memory. The default constructor starts on line 13. It takes no parameters. It calls the constructor defined on line 8 via the peculiar-looking *this(10)* call. The compiler will sort out which constructor *this()* refers to by examining the parameter list. Since there is a constructor defined to take an integer as a parameter, it will use that constructor.

Compile the code to ensure you didn't break anything. Then add some more methods so you can start seriously testing the PeopleManager class.

Defining Additional PeopleManager Methods

I recommend adding the capability to add Person objects to the people_array first. Then you can add the capability to list their information, and finally, the capability to delete them.

A good candidate name for a method that adds a person would be AddPerson(). Likewise, a good candidate name for a method that lists the Person objects in the array might be ListPeople(). Example 9.20 gives the source code for the PeopleManager class containing the newly created AddPerson() and ListPeople() methods.

9.20 PeopleManager.cs (4th Iteration)

```

1  using System;
2
3  public class PeopleManager {
4      // private fields
5      private Person[] people_array;
6      int index = 0;
7
8      // overloaded constructor
9      public PeopleManager(int length){
10         people_array = new Person[ length];
11     }
12
13     // default constructor
14     public PeopleManager():this(10){ }
15
16
17     public void AddPerson(String firstName, String middleName, String lastName,
18         Person.Sex gender, int dob_year, int dob_month, int dob_day){
19         if(index >= people_array.Length){
20             index = 0;
21         }
22         if(people_array[index] == null){
23             people_array[index++] = new Person(firstName, middleName, lastName, gender,
24                 new DateTime(dob_year, dob_month, dob_day));
25         }
26     } // end method
27
28
29     public void ListPeople(){
30         for(int i = 0; i<people_array.Length; i++){
31             if(people_array[i] != null){
32                 Console.WriteLine(people_array[i]);
33             }
34         }
35     } // end method
36 } // end PeopleManager class

```

Referring to Example 9.20 — let's look at the `AddPerson()` method for a moment. It has a parameter list that contains all the elements required to create a `Person` object. These include `firstName`, `middleName`, `lastName`, `gender`, `dob_year`, `dob_month`, and `dob_day`. The first thing the `AddPerson()` method does is to check to see if the value of the `index` field is greater than or equal to the length of the `people_array`. If so, it resets its value to 0. This guards against the possibility of exceeding the bounds of the array. Next, the `AddPerson()` method checks to see if a particular array element is equal to null. The first time the `AddPerson()` method is called on a particular `PeopleManager` object all the `people_array` elements will be null.

In this simple example, the `AddPerson()` method will add `Person` objects to the array until the array is full. From then on it will only insert `Person` objects if the array element it's trying to access is null. There are better ways to implement this method and they are left as exercises at the end of the chapter.

The `ListPerson()` method simply iterates over the `people_array`. If the array element is not null (meaning it points to a `Person` object) it uses that array element as an argument to the `WriteLine()` method, which in turn calls the `Person` object's `ToString()` method automatically.

TESTING THE PEOPLEMANAGER CLASS

You can use the `PeopleManagerApplication` class once again to test the functionality of the `PeopleManager` class. Example 9.21 gives the source code for the modified `PeopleManagerApplication` class. Figure 9-15 shows the results of running this program.

9.21 *PeopleManagerApplication.cs*
(Testing the *PeopleManager* class)

```

1  using System;
2
3  public class PeopleManagerApplication {
4      public static void Main(){
5          PeopleManager pm = new PeopleManager(); // default constructor call
6          pm.AddPerson("Jeff", "J", "Meyer", Person.Sex.MALE, 1975, 03, 12);
7          pm.AddPerson("Pete", "M", "Luongo", Person.Sex.MALE, 1967, 06, 18);
8          pm.AddPerson("Alex", "T", "Remily", Person.Sex.MALE, 1965, 11, 24);
9          pm.ListPeople();
10     } // end Main
11 } // end class definition

```

Referring to Example 9.21 — the `PeopleManager` default constructor is tested on line 5. This also tests the other `PeopleManager` constructor. Killed two birds with one stone here! The `AddPerson()` method is tested on lines 6 through 8, and the `ListPeople()` method is tested on line 9. Everything appears to work as expected. You can now add the capability to delete `Person` objects and perhaps some other functionality as well.

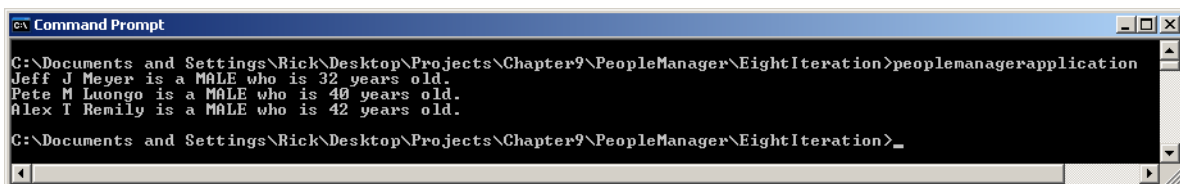


Figure 9-15: Results of Running Example 9.21

Adding FEATURES TO THE PEOPLEMANAGER CLASS

The `PeopleManager` class now implements two out of three required features. You can add `Person` objects to the `people_array` and you can list information about each `Person` object contained in the `people_array`. It's now time to implement the capability to delete `Person` objects from the array. A good candidate name for a method to delete a `Person` object from the array is `DeletePerson()`. See — method naming isn't so hard! But wait, not so fast. You just can't delete a `Person` from an arbitrary element. It might be better instead to delete a `Person` object from a specific `people_array` element, in which case you might want to better name the method `DeletePersonAtIndex()`.

While you're at it, you might want to add the capability to insert `Person` objects into a specific element within the array. A good candidate name for such a method might be `InsertPersonAtIndex()`. Example 9.22 gives the source code for the modified `PeopleManager` class.

9.22 PeopleManagerClass.cs
(5th Iteration)

```

1  using System;
2
3  public class PeopleManager {
4      // private fields
5      private Person[] people_array;
6      int index = 0;
7
8      // overloaded constructor
9      public PeopleManager(int length){
10         people_array = new Person[ length ];
11     }
12
13     // default constructor
14     public PeopleManager():this(10){ }
15
16     public void AddPerson(String firstName, String middleName, String lastName,
17         Person.Sex gender, int dob_year, int dob_month, int dob_day){
18         if(index >= people_array.Length){
19             index = 0;
20         }
21         if(people_array[ index ] == null){
22             people_array[ index++ ] = new Person(firstName, middleName, lastName, gender,
23                 new DateTime(dob_year, dob_month, dob_day));
24         }
25     } // end method
26
27     public void ListPeople(){
28         for(int i = 0; i<people_array.Length; i++){
29             if(people_array[ i ] != null){
30                 Console.WriteLine(people_array[ i ] );
31             }
32         }
33     } // end method
34
35
36     public void DeletePersonAtIndex(int index){
37         if(!(index < 0) || (index >= people_array.Length)){
38             people_array[ index ] = null;
39             this.index = index;
40         }
41     }
42
43     public void InsertPersonAtIndex( int index, String firstName, String middleName,
44         String lastName, Person.Sex gender, int dob_year,
45         int dob_month, int dob_day){
46         if(!(index < 0) || (index >= people_array.Length)){
47             this.index = index;
48             people_array[ this.index++ ] = new Person(firstName, middleName, lastName, gender,
49                 new DateTime(dob_year, dob_month, dob_day));
50         }
51     }
52
53 } // end PeopleManager class

```

Referring to Example 9.22 — examine closely for a moment the `DeletePersonAtIndex()` method whose definition starts on line 36. It declares one parameter named `index`. This parameter name will hide the field named `index` which is the desired behavior in this case. There is also a danger that the argument used in the `DeletePersonAtIndex()` method call might be invalid given the length of the `people_array`. The `if` statement on line 37 enforces the *precondition* that the value of the `index` parameter must be greater than or equal to zero or less than the length of the `people_array`. A similar test is made on the `index` parameter of the `InsertPersonAtIndex()`.

Example 9.23 gives the source code for the `PeopleManagerApplication` class that tests the newly added `PeopleManager` class functionality. Figure 9-16 shows the results of running this program.

9.23 PeopleManagerApplication.cs
(Testing new PeopleManager class functionality)

```

1  using System;
2
3  public class PeopleManagerApplication {
4      public static void Main(){
5          PeopleManager pm = new PeopleManager(); // default constructor call
6          pm.AddPerson("Jeff", "J", "Meyer", Person.Sex.MALE, 1975, 03, 12);
7          pm.AddPerson("Pete", "M", "Luongo", Person.Sex.MALE, 1967, 06, 18);
8          pm.AddPerson("Alex", "T", "Remily", Person.Sex.MALE, 1965, 11, 24);
9          pm.ListPeople();
10         Console.WriteLine("-----");
11         pm.DeletePersonAtIndex(0);

```

```

12     pm.ListPeople();
13     Console.WriteLine("-----");
14     pm.InsertPersonAtIndex(0, "Coralie", "S", "Miller", Person.Sex.FEMALE, 1963, 04, 04);
15     pm.ListPeople();
16 } // end Main
17 } // end class definition

```

Figure 9-16: Results of Running Example 9.23

Quick Review

The `PeopleManager` class implementation process followed the same pattern as that of class `Person`. It started with the class shell and added fields and methods as required to implement the necessary functionality. Develop code incrementally by applying the development cycle in an iterative fashion.

MORE ABOUT METHODS

In this section I'd like to focus your attention on several behavioral aspects of methods you will find helpful to fully understand before attempting more complex programming projects. You need to know the difference between *value parameters* and *reference parameters*, and be aware of local variable scoping rules.

VALUE PARAMETERS AND REFERENCE PARAMETERS

There are two ways to pass arguments to methods: 1) by *value*, or 2) by *reference*. A method parameter that omits the optional `ref` modifier is a value parameter by default. It's critical that you understand completely the difference between these two modes of parameter behavior or your methods may not work as you expect.

VALUE PARAMETERS: THE DEFAULT PARAMETER PASSING MODE

Two sorts of things can be passed as arguments to a method: 1) a value type object or 2) a reference that points to an object, otherwise simply referred to as a reference. When an argument is passed to a method, a *copy* of the argument is made and assigned to its associated method parameter. This is referred to as *pass by copy* or *pass by value*. I say again, behind the scenes, both categories of objects, value and reference, are copied into memory areas accessible to the method. Once copied, value types behave one way and reference types behave another way.

Consider for a moment the following method declaration:

```

public void SomeMethod(int int_param, Object object_ref_param){
    // body statements omitted
}

```

The `SomeMethod()` method declares two parameters: one value type `int` parameter and one `Object` reference parameter. This means that `SomeMethod()` can take two arguments: the first must be an integer and the second can be a reference to any `Object`. Remember — classes are reference types and structures are value types! Also remember that a reference contains a value that represents the memory location of the object to which it points. The values contained in these two arguments (an `int` and a reference) are copied to their corresponding parameters during the early

stages of the call to `SomeMethod()`. When `SomeMethod()` executes, it is only operating on its parameters, meaning it is only operating on copies of the original argument values.

For value types, this simply means that any change of value made to a method's parameter will only affect the copy — not the original value. The same holds true for reference parameters. A reference parameter will point to the same object the reference argument points to unless, during the method call, the reference parameter is changed to point to a different object. This change will only affect the parameter or copy — not the original reference used as an argument to the method call. Bear in mind, however, that as long as a reference parameter points to the same object the argument points to, changes to the object made via the parameter will have the same effect as though they were made via the argument itself. Figure 9-17 illustrates these concepts using a class's fields as method arguments.

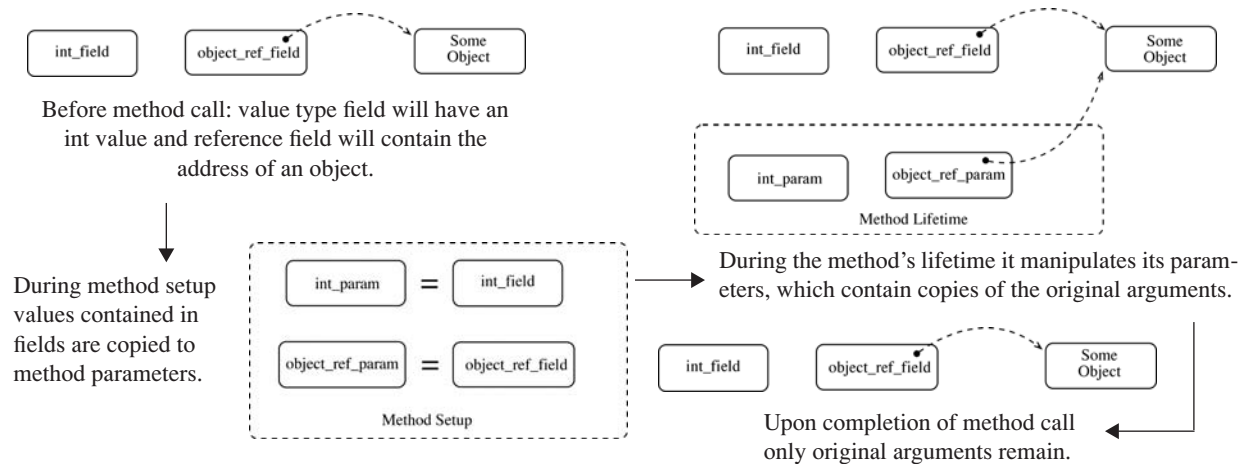


Figure 9-17: Default Value Parameter Behavior

Referring to Figure 9-17 — Prior to a method call, value type and reference fields contain values. During method setup these values are copied to their corresponding method parameters. The parameters can be manipulated by the method during the method's lifetime. Changes to the parameter values will only affect the parameters, not the original arguments. After the method call, value types and references used as arguments will retain their original values. Changes to the object pointed to by the reference parameter will remain in effect.

REFERENCE PARAMETERS: USING THE `ref` PARAMETER MODIFIER

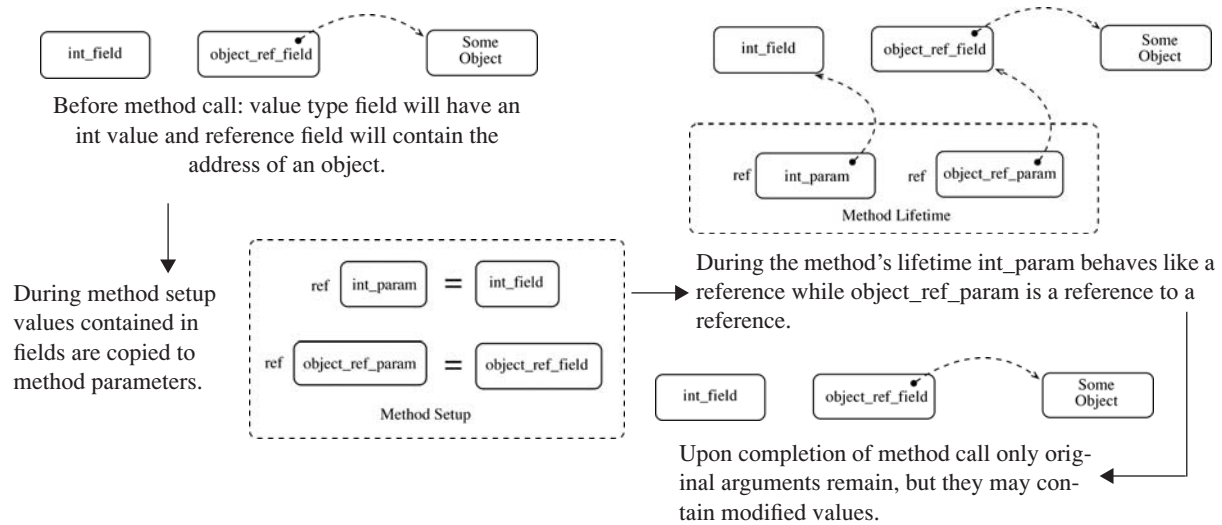
The `ref` modifier can be applied to parameters to change the way they behave inside of a method. Consider for a moment this modified version of `SomeMethod()`:

```
public void SomeMethod(ref int int_param, ref Object object_ref_param){
    // body statements omitted
}
```

In this version, the `ref` modifier is applied to each parameter. Figure 9-18 illustrates how these reference parameters behave differently from value parameters.

Referring to Figure 9-18 — the `ref` modifier changes the behavior of the method's parameters. Value type arguments like `int`, `float`, `double`, etc., behave as though they are references. Any change made to a value type `ref` parameter in the body of the method affects the original argument. In the case of reference type arguments, the `ref` parameter is a reference to a reference, as you can see from the diagram. What this means is that if you create a new object in the body of the method and assign its address to a reference parameter, it will be assigned to the original argument reference and it will now point to the new object.

I see that glazed look in your eyes. Check out the following two programs and note their behavior. Example 9.24 demonstrates the scenario shown in Figure 9-17. Example 9.25 demonstrates the scenario shown in Figure 9-18.

Figure 9-18: Reference Parameter Behavior — Using `ref` Modifier9.24 *ValueParameterTest.cs*

```

1  using System;
2  using System.Text;
3
4  public class ValueParameterTest {
5
6      int int_field;
7      StringBuilder object_ref_field = new StringBuilder();
8
9      public void F(int int_param, StringBuilder object_ref_param){
10         int_param = 2;
11         Console.WriteLine("Value of int_param modified in method: " + int_param);
12         object_ref_param.Append("Two");
13         Console.WriteLine("The value of object_ref_param after calling Append() in method: "
14             + object_ref_param);
15         object_ref_param = new StringBuilder();
16         object_ref_param.Append("Three");
17         Console.WriteLine("The value of object_ref_param after calling Append() in method: "
18             + object_ref_param);
19     }
20
21     public void G(){
22         int_field = 1;
23         object_ref_field.Append("One");
24         Console.WriteLine("The value of int_field before method call is: " + int_field);
25         Console.WriteLine("The value of the object_ref_field before method call is: " + object_ref_field);
26         Console.WriteLine("-----");
27         F(int_field, object_ref_field);
28         Console.WriteLine("-----");
29         Console.WriteLine("The value of int_field after method call is: " + int_field);
30         Console.WriteLine("The value of the object_ref_field after method call is: " + object_ref_field);
31     }
32
33 }
34
35 public static void Main(){
36     ValueParameterTest pt = new ValueParameterTest();
37     pt.G();
38 } // end Main
39 } // end class definition

```

Referring to Example 9.24 — the `ValueParameterTest` class declares two fields: `int_field` and `object_ref_field`. The `object_ref_field` is of type `StringBuilder`. The method `F()` whose definition begins on line 9 declares two parameters, one of type `int` named `int_param` and one of type `StringBuilder` named `object_ref_param`. The important thing to note in the body of method `F()` is that after the `Append()` method is called on the initial `object_ref_param` value, a new `StringBuilder` is created on line 15 and its reference is assigned to `object_ref_param`. Note that this has no effect on the reference value contained in `object_ref_field`.

Method `G()` whose definition begins on line 22 simply prints field values to the console before and after calling method `F()`. Again, value parameter passing is the default mode. Compare this code with Example 9.25.

9.25 RefParameterTest.cs

```

1  using System;
2  using System.Text;
3
4  public class RefParameterTest {
5
6      int int_field;
7      StringBuilder object_ref_field = new StringBuilder();
8
9      public void F(ref int int_param, ref StringBuilder object_ref_param){
10         int_param = 2;
11         Console.WriteLine("Value of int_param modified in method: " + int_param);
12         object_ref_param.Append("Two");
13         Console.WriteLine("The value of object_ref_param after calling Append() in method: "
14             + object_ref_param);
15         object_ref_param = new StringBuilder();
16         object_ref_param.Append("Three");
17         Console.WriteLine("The value of object_ref_param after calling Append() in method: "
18             + object_ref_param);
19     }
20
21     public void G(){
22         int_field = 1;
23         object_ref_field.Append("One");
24         Console.WriteLine("The value of int_field before method call is: " + int_field);
25         Console.WriteLine("The value of the object_ref_field before method call is: " + object_ref_field);
26         Console.WriteLine("-----");
27         F(ref int_field, ref object_ref_field);
28         Console.WriteLine("-----");
29         Console.WriteLine("The value of int_field after method call is: " + int_field);
30         Console.WriteLine("The value of the object_ref_field after method call is: " + object_ref_field);
31     }
32 }
33
34 public static void Main(){
35     RefParameterTest pt = new RefParameterTest();
36     pt.G();
37 } // end Main
38 } // end class definition

```

Referring to Example 9.25 — the differences between this code and the previous example is the following: 1) the class name, 2) the `ref` modifier has been applied to both of method `F()`'s parameters, and 3) the `ref` argument modifier has been applied to the arguments passed to the `F()` method call on line 28. This is required otherwise you will receive a compiler error. Figures 9.19 and 9.20 show the results of running these programs.

```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>valueparametertest
The value of int_field before method call is: 1
The value of the object_ref_field before method call is: One
-----
Value of int_param modified in method: 2
The value of object_ref_param after calling Append() in method: OneTwo
The value of object_ref_param after calling Append() in method: Three
-----
The value of int_field after method call is: 1
The value of the object_ref_field after method call is: OneTwo
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>_

```

Figure 9-19: Results of Running Example 9.24

```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>refparametertest
The value of int_field before method call is: 1
The value of the object_ref_field before method call is: One
-----
Value of int_param modified in method: 2
The value of object_ref_param after calling Append() in method: OneTwo
The value of object_ref_param after calling Append() in method: Three
-----
The value of int_field after method call is: 2
The value of the object_ref_field after method call is: Three
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\ParameterBehaviorTest>_

```

Figure 9-20: Results of Running Example 9.25

THE `out` PARAMETER MODIFIER

The `out` parameter modifier indicates that a parameter will be used to return a result to the calling program via its associated argument. An `out` parameter is similar to a `ref` parameter, but differs in that the initial value of an `out` parameter's associated argument is not important. Example 9.26 shows the use of the `out` parameter modifier.

9.26 *OutParamTest.cs*

```

1  using System;
2
3  public class OutParamTest {
4      int _a = 2;
5      int _count = 10;
6      long _result;
7
8      public void Factor(int value, int power, out long total){
9          total = 1;
10         for(int i = 1; i <= power; i++){
11             total = total * value;
12             Console.WriteLine("Value of i is {0} and value of total is {1}", i, total);
13         }
14     }
15
16     public void Run(){
17         Console.WriteLine("The value of _result before calling Factor is: " + _result);
18         Console.WriteLine("-----");
19         Factor(_a, _count, out _result);
20         Console.WriteLine("-----");
21         Console.WriteLine("The value of _result after calling Factor is: " + _result);
22     }
23
24     public static void Main(){
25         OutParamTest pt = new OutParamTest();
26         pt.Run();
27     } // end Main
28 } // end class definition

```

Referring to Example 9.26 — the class defines three fields: `_a`, `_count`, and `_result`. The `Factor()` method whose definition begins on line 8 declares three parameters, the last of which is an `out` parameter named `total`. The `Run()` method on line 16 writes the value of `_result` to the console before and after the `Factor()` method call. Figure 9-21 shows the results of running this program.

```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\OutParamTest>outparamtest
The value of _result before calling Factor is: 0
-----
Value of i is 1 and value of total is 2
Value of i is 2 and value of total is 4
Value of i is 3 and value of total is 8
Value of i is 4 and value of total is 16
Value of i is 5 and value of total is 32
Value of i is 6 and value of total is 64
Value of i is 7 and value of total is 128
Value of i is 8 and value of total is 256
Value of i is 9 and value of total is 512
Value of i is 10 and value of total is 1024
-----
The value of _result after calling Factor is: 1024
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\OutParamTest>

```

Figure 9-21: Results of Running Example 9.26

PARAMETER ARRAYS: USING THE `params` MODIFIER

Methods can take an indefinite number of arguments with the help of parameter arrays. Use the `params` modifier to declare an array parameter argument. If a parameter array appears in a method's parameter list, it must be either the last parameter in the list or the only parameter. Example 9.27 offers a short program that demonstrates the use of a parameter array.

9.27 *ParamArrayTest.cs*

```

1  using System;
2
3  public class ParamArrayTest {
4
5      public void ParamMethod(params String[] args){
6          Console.WriteLine("Method called with {0} arguments.", args.Length);
7          for(int i = 0; i < args.Length; i++){

```

```

8         Console.WriteLine("Argument " + i + " is " + args[i]);
9     }
10 }
11
12 public static void Main(){
13     ParamArrayTest pt = new ParamArrayTest();
14     pt.ParamMethod();
15     pt.ParamMethod("one");
16     pt.ParamMethod("one", "two");
17     pt.ParamMethod(new String[] { "one", "two", "three" });
18 }
19 }

```

Referring to Example 9.27 — the `ParamMethod()` whose definition begins on line 5 declares a `String` parameter array. The method simply prints the number of arguments it was called with, and then prints the value of each argument to the console. The method's use is demonstrated in the `Main()` method. The `ParamMethod()` is called on line 14 with no arguments, followed by a call with one argument, next with two arguments, then finally with a `String` array that contains three arguments. I included this last method call to show you how arguments can also be passed as an array of the type expected by the method. Figure 9-22 shows the results of running this program.

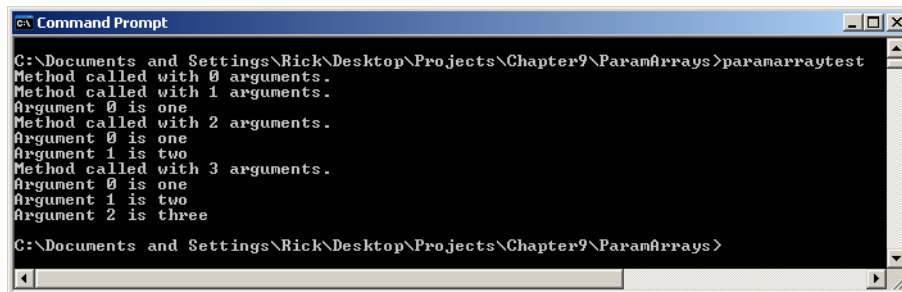


Figure 9-22: Results of Running Example 9.27

LOCAL VARIABLE SCOPING

Methods can declare variables for use within the method body. These variables are known as *local variables*. The scope of a local variable includes the method body block or code block in which it is declared, however, it is only available for use after its point of declaration. Parameters are considered to be local variables and are available for use from the beginning to the end of the method body.

A local variable whose name is the same as a class or instance field will hide that field from the method body. To access the field you must preface its name with the *this* keyword. Or, better still, change the field's name or the local variable's name to eliminate the problem!

ANYWHERE AN OBJECT OF <type> IS REQUIRED, A METHOD THAT RETURNS <type> CAN BE USED

The title of this section says it all. Anywhere an object of a certain *type* is required, a method that returns a result of that *type* can be used. Substitute the word *type* in the previous sentence for any value or reference type you require. For reference types, the *new* keyword can be used to create argument objects on the fly. Refer to the following method declaration once again:

```

public void SomeMethod(int int_param, Object object_ref_param){
    // body statements omitted
}

```

Assume for this example that the following fields and methods exist as well: `int_field`, `object_reference_field`, `GetInt()` and `GetObject()`. Assume for this example that `GetInt()` returns an `int` value and that `GetObject()` returns a reference to an `Object`. Given these fields and methods the `SomeMethod()` could be called in the following ways:

```

SomeMethod(int_field, object_reference_field);
SomeMethod(GetInt(), object_reference_field);
SomeMethod(int_field, GetObject());
SomeMethod(GetInt(), GetObject());
SomeMethod(GetInt(), new Object());

```

As you progress through this book and your knowledge of C# grows, you will be exposed to all the above forms of a method call plus several more.

Quick Review

By default, arguments are passed to a method call by value. This is also referred to as *pass by copy*. The method parameters contain a copy of the argument values. Any change to the parameter values only affect the copies, not the actual arguments. Changes to an object pointed to by a reference parameter will affect the original object. However, a change to what a reference parameter points to only affects the parameter, not the original reference argument.

Use the `ref` parameter modifier to change parameter behavior so that changes made to the parameters also affect the original argument values. By using the `ref` parameter, a change to what a reference parameter points to (*i.e.*, creating a new object with the new operator) will also change what the original reference argument points to.

Use the `out` parameter modifier if you need to return method results via one or more of its arguments.

Use the `params` modifier to create parameter arrays.

Methods can contain local variables whose scope is the body code block or the code block in which they are declared. Local variables are available for use after the point of their declaration up to the end of the code block. Method parameters are local variables that are available to the entire method body.

Anywhere an object of `<type>` is required, a method that returns that `<type>` can be used in its place.

STRUCTURES VS. CLASSES

Structures (structs) share many similarities with classes with one huge difference; a structure defines a new value type whereas a class defines a new reference type, as is shown in Figure 9-23. This section highlights the differences between structures and classes and offers some advice on when you might want to use a structure vs. a class.

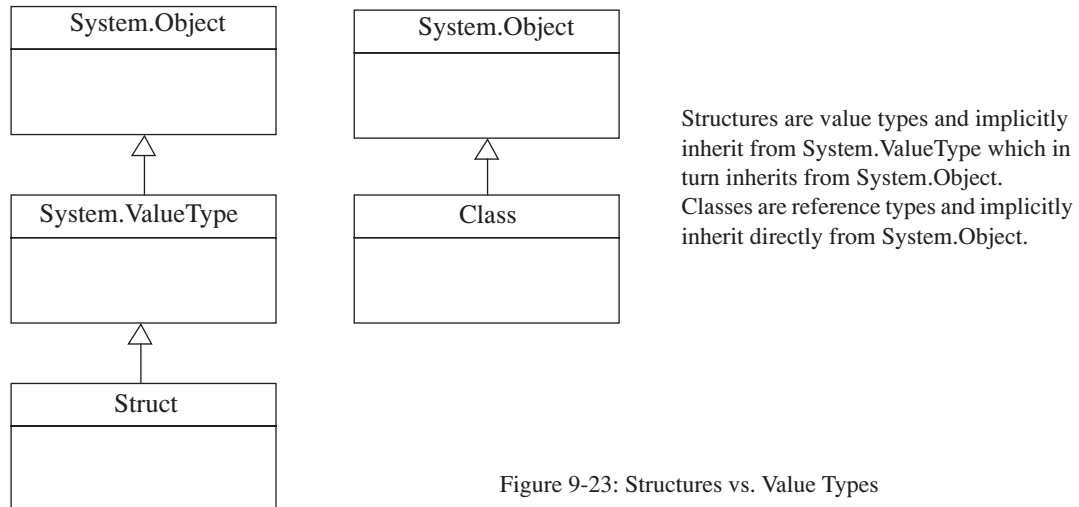


Figure 9-23: Structures vs. Value Types

VALUE SEMANTICS VS. REFERENCE SEMANTICS

A structure type (value type) variable directly contains the data associated with the structure as opposed to a class type (reference type) variable that contains a reference to an object in memory. It is possible for two different reference variables to point to the same object in memory. But not so for value type variables, where each variable has its own copy of the data.

Structures are not allocated on the heap unless they undergo a *boxing* operation. Boxing and unboxing are covered below.

Because structure variables are not reference variables they cannot be null.

TEN AUTHORIZED MEMBERS VS. ELEVEN

Structures can have *constants, fields, methods, properties, events, indexers, operators, constructors, static constructors, and nested type declarations.*

A structure cannot have a finalizer, nor can you define an explicit parameterless (default) constructor.

DEFAULT VARIABLE FIELD VALUES

As stated above, you cannot define a parameterless (default) constructor for a structure. The compiler-supplied default constructor will set all a structure's value type fields to their default values and any reference type fields to null. Also, you cannot use instance field initializers to set the values of each field.

BEHAVIOR DURING ASSIGNMENT

The assignment of one value type variable to another causes a *complete copy of the structure's data* being assigned. Compare this behavior to that of a reference type where only the *reference to an object* is copied from the variable being assigned. Recall that when value types are passed as arguments to methods, a copy of the argument is assigned to its corresponding parameter. This behavior was discussed in detail earlier in this chapter. (*Also, see Chapter 22 — Well-Behaved Objects*)

this BEHAVES DIFFERENTLY

In a structure, `this` is considered a variable via which the values of the structure can be assigned to and modified. In an instance constructor, `this` functions like an `out` parameter. In an instance method, `this` functions like a `ref` parameter.

INHERITANCE NOT ALLOWED

You cannot extend a structure. Structures are never abstract and always inherently sealed. Structures can implement interfaces but they cannot specify a base class. Structure members cannot be abstract or virtual. The `override` keyword is only allowed when overriding members of `System.ValueType`.

BOXING AND UNBOXING

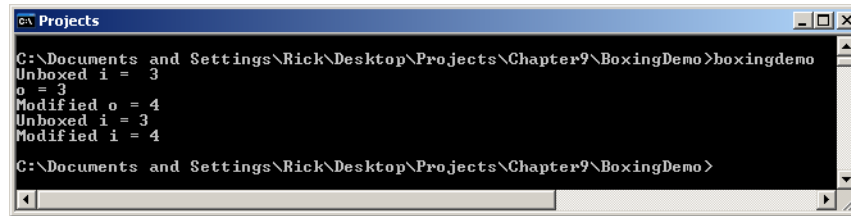
If you need to treat a value type like a reference type, you can *box* the value-type into an object that is then allocated on the heap. Look at the following example.

9.28 *BoxingDemo.cs*

```

1  using System;
2
3  public class BoxingDemo {
4
5      public static void Main(){
6          int i = 3;
7          Console.WriteLine("Unboxed i = " + i);
8          object o = i; // boxing
9          Console.WriteLine("o = " + o);
10         o = 4;        // treat o like an int
11         Console.WriteLine("Modified o = " + o);
12         Console.WriteLine("Unboxed i = " + i);
13         i = (int)o;    // unboxing
14         Console.WriteLine("Modified i = " + i);
15     }
16 }
```

Referring to Example 9.28 — the local variable `i` is declared and initialized to the value 3. On line 8, an object reference named `o` is declared and the value-type `i` is boxed by the assignment to `o`. The reference `o` now points to a boxed integer value type. Line 10 demonstrates that assignments to `o` can take place like assignments to ordinary integers. On line 13, the value type contained in `o` is unboxed and assigned to the variable `i`. The explicit cast is required. Figure 9-24 shows the results of running this program.



```

C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\BoxingDemo>boxingdemo
Unboxed i = 3
o = 3
Modified o = 4
Unboxed i = 3
Modified i = 4
C:\Documents and Settings\Rick\Desktop\Projects\Chapter9\BoxingDemo>

```

Figure 9-24: Results of Running Example 9.28

WHEN TO USE STRUCTURES

Structures are appropriate when the amount of data they contain is small. Remember, when one structure variable is assigned to another a complete copy of the assigned structure's data is copied over. The ultimate answer to the structure vs. class question can only be answered by thoroughly accessing a project's design and performance requirements.

SUMMARY

Problem abstraction requires lots of programmer creativity and represents the *art* in the *art of programming*. Your guiding mantra during problem abstraction is to *amplify the essential, eliminate the irrelevant*. Problem abstraction is performed in the analysis and design phase of the development cycle. The abstractions you choose to model a particular problem will directly influence a program's design.

The end result of problem abstraction is the identification and creation of one or more new data types. The data types derived through problem abstraction are referred to as *abstract data types* (ADTs) or *user-defined data types*. User-defined data types can be implemented as *structures* or *classes*. These structures or classes will interact with each other in some capacity to implement the complete problem solution.

A UML class diagram shows the static relationship between classes that participate in a software design. Programmers use the class diagram to express and clarify design concepts to themselves, to other programmers, to management, and to clients.

In UML, a rectangle represents a class. The rectangle can have three compartments. The uppermost compartment contains the class name, the middle compartment contains fields, and the bottom compartment contains the methods.

A stereotype introduces a new type of element within a system. The stereotype name is contained within the guillemet characters << >>.

Generalization and *specialization* are indicated by lines tipped with hollow arrows. The arrow points from the specialized class to the generalized class. The generalized class is the base class, and the specialized class is the derived or subclass. Generalizations specify "is a..." relationships between base and subclasses.

Dependencies are indicated by dashed arrows pointing to the class being depended upon. Dependencies are one way to indicate "uses..." relationships between classes.

C# classes can contain eleven different types of members: *fields*, *constants*, *methods*, *properties*, *events*, *indexers*, *operators*, *instance constructors*, *static constructors*, *finalizers*, and *nested type declarations*.

The access modifiers `public`, `protected`, `private`, `internal`, and `protected internal` are used to control access to class and instance members. If no access is specified then `private` is assumed.

The term *horizontal access* describes the access a client object has to the members of a server object. The client object represents the code that uses the services of another object. It can do this in two ways: 1) by accessing a class's public static members via the class name, or 2) by creating an instance of the class and accessing its public non-static members via an object reference.

Methods are named modules of executable program functionality. Methods contain program statements that, when grouped together, represent a basic level of code reuse. You access the functionality of a method by calling the method using its name in a program.

Methods should be *well named* and *maximally cohesive*. A well named, maximally cohesive method will pull no surprises!

Method definitions have structure. Their behavior can be optionally modified with method modifiers, they can optionally specify a return result type or `void`, and they can have an optional parameter list.

Methods have a distinguishing characteristic known as a *method signature*. Methods with different names and parameter lists are said to have different signatures. Methods with different names and the same parameter list also have different signatures. Methods with the same name and different parameter lists have different signatures as well and are said to be *overloaded* (because they share the same name). Methods cannot have the same name and identical parameter lists. This will cause a compiler error.

Constructor methods set up or build an object when it's created in memory. If you do not provide one, the compiler will create a default constructor for you, but it may or may not provide the level of functionality you require.

Incrementally build and test abstract data types by iteratively applying the steps of the development cycle. Start with the class definition shell and then add fields, properties, and methods as required to fulfill the class's design objectives.

Test class functionality with the help of a *test driver*. A test driver is a small program that's used to exercise the functionality of another program.

By default, arguments are passed to a method call by value. This is also referred to as *pass by copy*. The method parameters contain a copy of the argument values. Any change to the parameter values only affect the copies, not the actual arguments. Changes to an object pointed to by a reference parameter will affect the original object. However, a change to what a reference parameter points to only affects the parameter, not the original reference argument.

Use the `ref` parameter modifier to change parameter behavior so that changes made to the parameters also affect the original argument values. By using the `ref` parameter, a change to what a reference parameter points to (*i.e.*, creating a new object with the `new` operator) will also change what the original reference argument points to.

Use the `out` parameter modifier if you need to return method results via one or more of its arguments.

Use the `params` modifier to create parameter arrays.

Methods can contain local variables whose scope is the body code block or the code block in which they are declared. Local variables are available for use after the point of their declaration up to the end of the code block. Method parameters are local variables that are available to the entire method body.

Anywhere an object of `<type>` is required, a method that returns that `<type>` can be used in its place.

Skill-Building Exercises

1. **.NET API Drill:** Browse through the .NET API and look for classes that contain static class methods or fields. Note how they are being used in each class.
2. **Problem Abstraction Drill:** Revisit the Robot Rat project presented in Chapter 3. Study the project specification and identify candidate classes. Make a table of the classes and list their names along with a description of their potential fields and functionality. Try not to be influenced by the solution approach taken in Chapter 3. Instead, focus on breaking the problem into potential classes and assigning functionality to those classes. For example, the program written in Chapter 3 is included in one large application class. At a minimum you will want to have a separate application class. Draw a UML diagram to express your design.
3. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of an automobile in code. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
4. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of an airplane in code. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
5. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a nuclear submarine. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.

6. **Further Research:** Research the topic of data encapsulation. The goal of your research should be to understand the role design plays in determining the level of data encapsulation and what design and programming strategies you can use to enforce data encapsulation.
7. **Coding Exercise:** Write a program that lets you experiment with the effects of method parameter passing. The names of the class and any fields and methods required are left to your discretion. The idea is to create a class that contains several value type and reference fields. It should also contain several methods that return value types and references. Write a method that takes at least one value type and one reference type parameter. Practice calling the method using a combination of fields, methods, and the `new` operator. Manipulate the parameters in the body of the method and note the results. Change the method's parameter behavior with the use of the `ref` and `out` modifiers. Again, write some code that calls the method and note the results on both the arguments supplied to the method and to the method's parameters during the method's lifetime.
8. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a computer in code. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
9. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a coffee maker. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.
10. **Problem Abstraction Drill:** Consider the problem of modeling the functionality of a gasoline pump. Create a list of candidate classes and include potential fields and methods. Draw a UML diagram to express your design.

SUGGESTED PROJECTS

1. **Improve the `PeopleManager.AddPerson()` Method:** Improve the functionality of the `AddPerson()` method of the `PeopleManager` class presented in this chapter. In its current state, the `AddPerson()` method only creates a new `Person` object and assigns the reference to the array element if the array element is null. Otherwise it does nothing and gives no indication that the creation and insertion of a new `Person` object failed. Make the following modifications to the `AddPerson()` method:
 - a. Search the `people_array` for a null element and insert the new `Person` reference at that element.
 - b. If the array is full, create a new array that's 1.5 times the size of the current `people_array` and then insert the new `Person` reference at that element.
 - c. Have the `AddPerson()` method return a boolean value indicating success or failure of the new `Person` object creation and insertion operation.
2. **Write a Submarine Commander Program:** Using the results of the problem abstraction performed in skill-building exercise 5, write a program that lets you create a fleet of nuclear submarines. You should be able to add submarines to the fleet, remove them from the fleet, and list all the submarines in your fleet. You will want to power-up their nuclear reactors and shut down their nuclear reactors. You will also want to fire their weapons. To keep this programming exercise manageable, just write simple messages to the console in response to commands sent to each submarine object.
3. **Write a Gasoline Pump Operation Program:** Using the results of the analysis you performed in skill-building exercise 10, write a program that lets you control the operation of a gasoline pump. You should be able to turn the gas pump on and off. You should only be able to pump gas when the pump is on. When you are done pumping gas, indicate how much gas was pumped in gallons or liters and give the total price of the gas pumped. Provide a way to set the price of gas.
4. **Write a Calculator Program:** Write a program that implements the functionality of a simple calculator. The focus of this project should be the creation of a class that performs the calculator's operations. Give the `Calculator` class

the ability to add, subtract, multiply, and divide integers and floating point numbers. Some of the Calculator class methods may need to be overloaded to handle different types of arguments.

5. **Write a Library Manager Program:** Write a program that lets you catalog the books in your personal library. The Book class should have the following attributes: title, author, and International Standard Book Number (ISBN). You can add any other attributes you deem necessary. Create a class named LibraryManager that lets you create and add books to your library, delete books from your library, and list the books in your library. Use an array to hold the books in your library. Research sorting routines and implement a SortBooks() method.
6. **Write a Linked-List Program:** A special property of C# classes is that the name of the class you are defining can be used to declare fields within that class. Consider the following code example:

9.29 Node.cs (Partial Listing)

```

1  public class Node {
2      private Node previous = null;
3      private Node next = null;
4      private Object payload = null;
5
6      // methods omitted for now
7  }

```

Here, the class name Node appears in the body of the Node class definition to declare two Node references named previous and next. This technique creates data structures designed for use within a linked list. Use the code shown in Example 9.29 to help you write a program that manages a linked list. Here are a few hints to get you started:

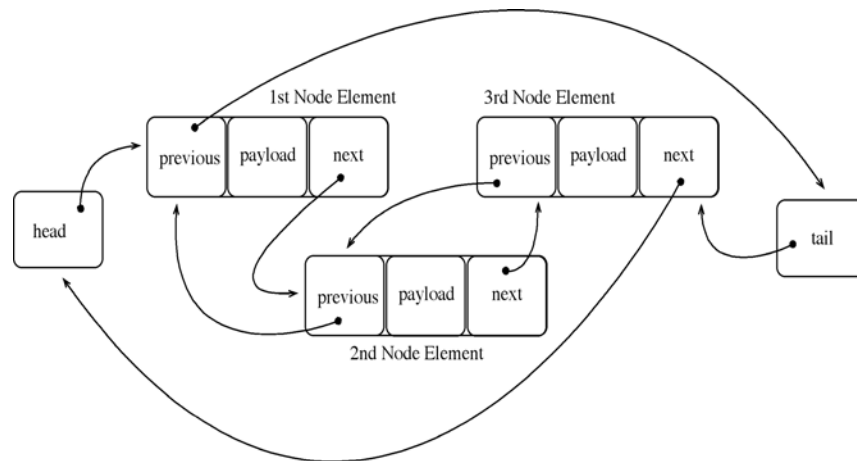


Figure 9-25: Circular Linked List with Three Nodes

Referring to Figure 9-25 above — a linked list contains one or more nodes and a head and a tail. The head points to the first node in the list. The tail always points to the last node in the list. Each node has a next and previous attribute along with a payload. Figure 9-25 shows a linked list having three nodes. The first node element's next attribute points to the second node element, and the second node element's next attribute points to the third node element. The third node element is the last node in the list and its next attribute points to the head, which always points to the first node in the list. Each node's previous attribute works in the opposite fashion. Because each node has a next and a previous attribute, it can be used to create circular linked list as is shown in Figure 9-25.

For this project, write a linked list manager program that lets you add, delete, and list the contents of each node in the list. You will have to add methods or properties to the Node class code given in Example 9.29. At a minimum you should add properties for each field.

This is a challenging project and will require you to put some thought into the design of both the Node and the LinkedListManager class. The most complicated part of the design will be figuring out how to insert and delete nodes into and from the list. When you successfully complete this project, you will have a good, practical understanding of references and how they are related to pointers in other programming languages.

7. **Convert The Library Manager To Use A Linked List:** Rewrite the library manager program presented in suggested project 6 to use a linked list of books instead of an array.

Self-Test Questions

1. Define the term problem abstraction. Explain why problem abstraction requires creativity.
2. What is the end result of problem abstraction?
3. Describe in your own words how you would go about the problem abstraction process for a typical programming problem.
4. What is the purpose of the UML class diagram? What geometric shape is used to depict classes in a UML class diagram? Where are class names, fields, and methods depicted on a class symbol?
5. What do the lines tipped with hollow arrowheads depict in a UML class diagram?
6. What are the eleven categories of C# class members?
7. What's the difference between static and non-static fields?
8. What the difference between static and non-static methods?
9. What's the difference between `readonly` fields and `const` fields?
10. List and describe the purpose of member access modifiers.
11. Explain the concept of horizontal access. Draw a picture showing how client code access to a server class's members is controlled using the access modifiers `public` and `private`.
12. What is a method?
13. List and describe the desirable characteristics of a method.
14. Explain the concept of *cohesion* as it pertains to methods.
15. (True/False) A method should be maximally cohesive.
16. What steps can you take as a programmer to ensure your methods are maximally cohesive?
17. What's the purpose of a method definition?
18. What parts of a method definition are optional?
19. What is meant by the term method signature?
20. What parts of a method are included in a method's signature?
21. What constitutes an overloaded method?
22. Give at least one example for which method overloading is useful.

23. What makes constructor methods different from ordinary methods?
24. Describe in your own words how arguments are passed to methods.

REFERENCES

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [<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>]

Rick Miller. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Second Edition. Addison-Wesley. Reading MA. ISBN: 0-201-89685-0

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

Sinan Si Alhir. *UML In A Nutshell: A Desktop Quick Reference*. O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 1-56592-448-7

NOTES
