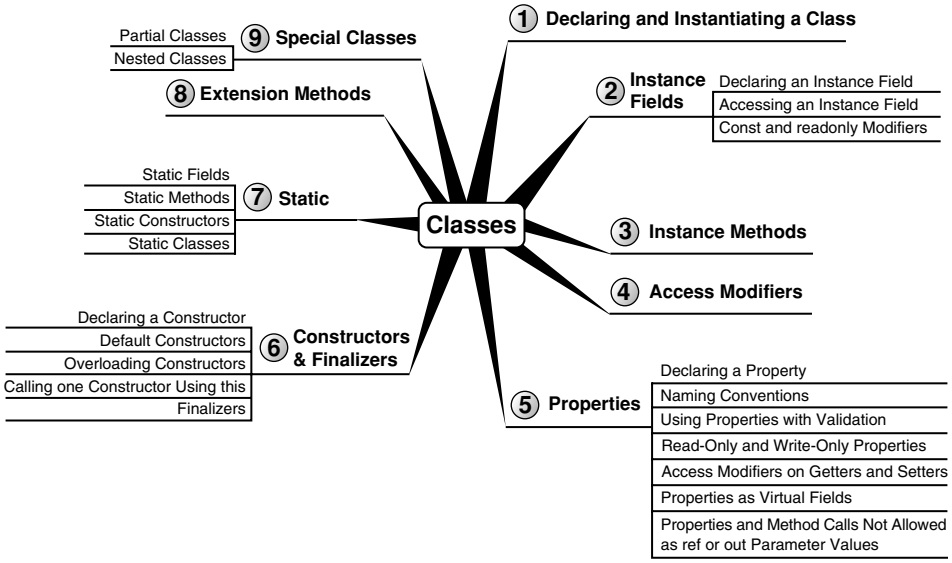
5
Classes

YOU BRIEFLY SAW IN CHAPTER 1 how to declare a new class called `HelloWorld`. In Chapter 2, you learned about the built-in primitive types included with C#. Since you have now also learned about control flow and how to declare methods, it is time to discuss defining your own types. This is the core construct of any C# program; this support for classes and the objects created from them is what makes C# an object-oriented language.



This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

This chapter introduces the basics of object-oriented programming using C#. A key focus is on how to define **classes,** which are the templates for objects themselves.

All of the constructs of structured programming from the previous chapters still apply within object-oriented programming. However, by wrapping those constructs within classes, you can create larger, more organized programs that are more maintainable. The transition from structured, control-flow-based programs to object-oriented programs revolutionized programming because it provided an extra level of organization. The result was that smaller programs were simplified somewhat. Even more importantly, it was easier to create much larger programs because the code within those programs was better organized.

One of the key advantages of object-oriented programming is that instead of creating new programs entirely from scratch, you can assemble a collection of existing objects from prior work, extending the classes with new features, adding more classes, and thereby providing new functionality.

Readers unfamiliar with object-oriented programming should read the Beginner Topic blocks for an introduction. The general text outside the Beginner Topics focuses on using C# for object-oriented programming with the assumption that readers are already familiar with object-oriented concepts.

This chapter delves into how C# supports encapsulation through its support of constructs such as classes, properties, and access modifiers; we covered methods in the preceding chapter. The next chapter builds on this foundation with the introduction of inheritance and the polymorphism that object-oriented programming enables.

BEGINNER TOPIC

Object-Oriented Programming

The key to programming successfully today lies in the ability to provide organization and structure to the implementation of the complex requirements of large applications. Object-oriented programming provides one of the key methodologies in accomplishing this goal, to the point that it is difficult for object-oriented programmers to envision transitioning back to structured programming, except for the most trivial programs.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

The most fundamental construct in object-oriented programming is the class. A group of classes form a programming abstraction, model, or template of what is often a real-world concept. The class `OpticalStorageMedia`, for example, may have an `Eject()` method on it that causes a disk to eject from the player. The `OpticalStorageMedia` class is the programming abstraction of the real-world object of a CD or DVD player.

Classes exhibit the three principal characteristics of object-oriented programming: encapsulation, inheritance, and polymorphism.

Encapsulation

Encapsulation allows you to hide details. The details can still be accessed when necessary, but by intelligently encapsulating the details, large programs are made easier to understand, data is protected from inadvertent modification, and code becomes easier to maintain because the effects of a code change are limited to the scope of the encapsulation. Methods are examples of encapsulation. Although it is possible to take the code from a method and embed it directly inline with the caller's code, refactoring of code into a method provides encapsulation benefits.

Inheritance

Consider the following example: A DVD drive is a type of optical media device. It has a specific storage capacity along with the ability to hold a digital movie. A CD drive is also a type of optical media device, but it has different characteristics. The copy protection on CDs is different from DVD copy protection, and the storage capacity is different as well. Both CD drives and DVD drives are different from hard drives, USB drives, and floppy drives (remember those?). All fit into the category of storage devices, but each has special characteristics, even for fundamental functions such as the supported filesystems and whether instances of the media are read-only or read/write.

Inheritance in object-oriented programming allows you to form "is a kind of" relationships between these similar but different items. It is reasonable to say that a DVD drive "is a kind of" storage media and that a CD drive "is a kind of" storage media, and as such, that each has storage capacity. We could also reasonably say that both have an "is a kind of" relationship with "optical storage media," which in turn "is a kind of" storage media.

If you define classes corresponding to each type of storage device mentioned, you will have defined a **class hierarchy,** which is a series of "is a

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

kind of" relationships. The base class, from which all storage devices derive, could be the class `StorageMedia`. As such, classes that represent CD drives, DVD drives, hard drives, USB drives, and floppy drives are derived from the class `StorageMedia`. However, the classes for CD and DVD drives don't need to derive from `StorageMedia` directly. Instead, they can derive from an intermediate class, `OpticalStorageMedia`. You can view the class hierarchy graphically using a Unified Modeling Language (UML)–like class diagram, as shown in Figure 5.1.

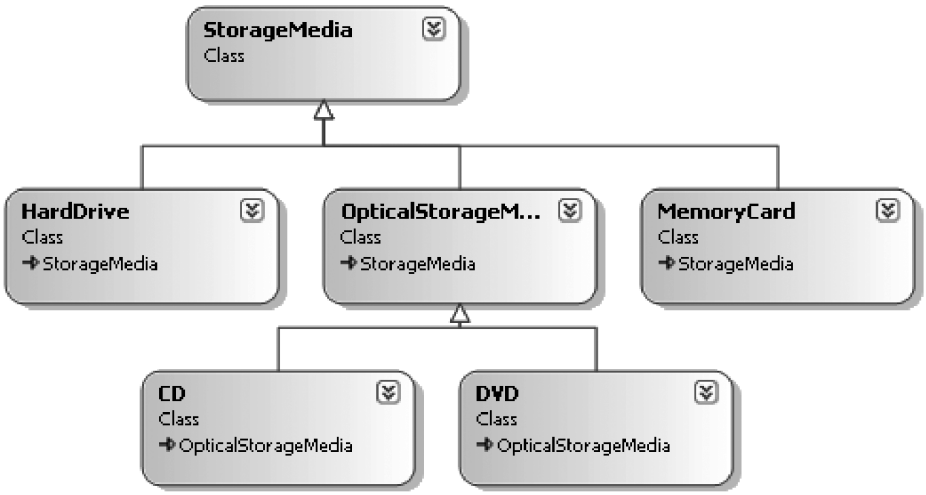


FIGURE 5.1: Class Hierarchy

The inheritance relationship involves a minimum of two classes, such that one class is a more specific kind of the other; in Figure 5.1, `HardDrive` is a more specific kind of `StorageMedia`. Although the more specialized type, `HardDrive`, is a kind of `StorageMedia`, the reverse is not true—that is, an instance of `StorageMedia` is not necessarily a `HardDrive`. As Figure 5.1 shows, inheritance can involve more than two classes.

The more specialized type is called the **derived type** or the **subtype.** The more general type is called the **base type** or the **super type.** The base type is also often called the "parent" type and its derived types are often called its "child" types. Though this usage is common, it can be confusing: After all, a child is not a kind of parent! In this book we will stick to "derived type" and "base type."

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

To **derive** or **inherit** from another type is to **specialize** that type, which means to customize the base type so that it is more suitable for a specific purpose. The base type may contain those implementation details that are common to all of the derived types.

The key feature of inheritance is that all derived types inherit the members of the base type. Often, the implementation of the base members can be modified, but regardless, the derived type contains the base type's members in addition to any other members that the derived type contains explicitly.

Derived types allow you to organize your classes into a coherent hierarchy where the derived types have greater specificity than their base types.

Polymorphism

Polymorphism is formed from *poly*, meaning "many," and *morph*, meaning "form." In the context of objects, polymorphism means that a single method or type can have many forms of implementation.

Suppose you have a media player that can play both CD music discs and DVDs containing MP3s. However, the exact implementation of the `Play()` method will vary depending on the media type. Calling `Play()` on an object representing a music CD or on an object representing a music DVD will play music in both cases, because each object's type understands the intricacies of playing. All that the media player knows about is the common base type, `OpticalStorageMedia`, and the fact that it defines the `Play()` method. Polymorphism is the principle that a type can take care of the exact details of a method's implementation because the method appears on multiple derived types, each of which shares a common base type (or interface) that also contains the same method signature.

Declaring and Instantiating a Class

Defining a class involves first specifying the keyword `class`, followed by an identifier, as shown in Listing 5.1.

LISTING 5.1: Defining a Class

```
class Employee
{
}
```

All code that belongs to the class will appear between the curly braces following the class declaration. Although not a requirement, generally you

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

place each class into its own file. This makes it easier to find the code that defines a particular class, because the convention is to name the file using the class name.

Guidelines

DO NOT place more than one class in a single source file.

DO name the source file with the name of the public type it contains.

Once you have defined a new class, you can use that class as though it were built into the framework. In other words, you can declare a variable of that type or define a method that takes a parameter of the new class type. Listing 5.2 demonstrates such declarations.

LISTING 5.2: Declaring Variables of the Class Type

```
class Program
{
  static void Main()
  {
      Employee employee1, employee2;
      // ...
  }

  static void IncreaseSalary(Employee employee)
  {
      // ...
  }
}
```

BEGINNER TOPIC

Objects and Classes Defined

In casual conversation, the terms *class* and *object* appear interchangeably. However, object and class have distinct meanings. A **class** is a template for what an object will look like at instantiation time. An **object,** therefore, is an instance of a class. Classes are like the mold for what a widget will look like. Objects correspond to widgets created by the mold. The process of creating an object from a class is called **instantiation** because an object is an instance of a class.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Now that you have defined a new class type, it is time to instantiate an object of that type. Mimicking its predecessors, C# uses the `new` keyword to instantiate an object (see Listing 5.3).

LISTING 5.3: Instantiating a Class

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      Employee employee2;
      employee2 = new Employee();

      IncreaseSalary(employee1);
  }
}
```

Not surprisingly, the assignment can occur in the same statement as the declaration, or in a separate statement.

Unlike the primitive types you have worked with so far, there is no literal way to specify an `Employee`. Instead, the `new` operator provides an instruction to the runtime to allocate memory for an `Employee` object, instantiate the object, and return a reference to the instance.

Although an explicit operator for allocating memory exists, there is no such operator for de-allocating the memory. Instead, the runtime automatically reclaims the memory sometime after the object becomes inaccessible. The **garbage collector** is responsible for the automatic de-allocation. It determines which objects are no longer referenced by other active objects and then de-allocates the memory for those objects. The result is that there is no compile-time–determined program location where the memory will be collected and restored to the system.

In this trivial example, no explicit data or methods are associated with an `Employee`, which renders the object essentially useless. The next section focuses on adding data to an object.

BEGINNER TOPIC

Encapsulation Part 1: Objects Group Data with Methods

If you received a stack of index cards with employees' first names, a stack of index cards with their last names, and a stack of index cards with their

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

salaries, the cards would be of little value unless you knew that the cards were in the same order in each stack. Even so, the data would be difficult to work with because determining a person's full name would require searching through two stacks. Worse, if you dropped one of the stacks, there would be no way to reassociate the first name with the last name and the salary. Instead, you would need one stack of employee cards in which all of the data is grouped on one card. With this approach, first names, last names, and salaries will be encapsulated together.

Outside the object-oriented programming context, to **encapsulate** a set of items is to enclose those items within a capsule. Similarly, object-oriented programming encapsulates methods and data together into an object. This provides a grouping of all of the class **members** (the data and methods within a class) so that they no longer need to be handled individually. Instead of passing a first name, a last name, and a salary as three separate parameters to a method, objects enable a call to pass a reference to an employee object. Once the called method receives the object reference, it can send a message (it can call a method such as AdjustSalary(), for example) to the object to perform a particular operation.

Language Contrast: C++—`delete` Operator

C# programmers should view the `new` operator as a call to instantiate an object, not as a call to allocate memory. Both objects allocated on the heap and objects allocated on the stack support the `new` operator, emphasizing the point that `new` is not about how memory allocation should take place and whether de-allocation is necessary.

Thus C# does not need the `delete` operator found in C++. Memory allocation and de-allocation are details that the runtime manages, allowing the developer to focus more on domain logic. However, though memory is managed by the runtime, the runtime does not manage other resources such as database connections, network ports, and so on. Unlike C++, C# does not support **implicit deterministic resource cleanup** (the occurrence of implicit object destruction at a compile-time–defined location in the code). Fortunately, C# does support **explicit deterministic resource cleanup** via a `using` statement, and **implicit nondeterministic resource cleanup** using finalizers.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Instance Fields

One of the key aspects of object-oriented design is the grouping of data to provide structure. This section discusses how to add data to the `Employee` class. The general object-oriented term for a variable that stores data within a class is **member variable.** This term is well understood in C#, but the more standard term and the one used in the specification is **field,** which is a named unit of storage associated with the containing type. **Instance fields** are variables declared at the class level to store data associated with an object. Hence, **association** is the relationship between the field data type and the containing field.

Declaring an Instance Field

In Listing 5.4, the class `Employee` has been modified to include three fields: `FirstName`, `LastName`, and `Salary`.

LISTING 5.4: Declaring Fields

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;
}
```

With these fields added, it is possible to store some fundamental data with every `Employee` instance. In this case, you prefix the fields with an access modifier of `public`. The use of `public` on a field indicates that the data within the field is accessible from classes other than `Employee` (see the section "Access Modifiers," later in this chapter).

As with local variable declarations, a field declaration includes the data type to which the field refers. Furthermore, it is possible to assign fields an initial value at declaration time, as demonstrated with the `Salary` field in Listing 5.5.

LISTING 5.5: Setting Initial Values of Fields at Declaration Time

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary = "Not enough";
}
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

We delay the guidelines of naming and coding fields until later in the chapter, after C# properties have been introduced. Suffice it to say, Listing 5.5 does *not* follow the general convention.

Accessing an Instance Field

You can set and retrieve the data within fields. However, the fact that a field does not include a `static` modifier indicates that it is an instance field. You can access an instance field only from an instance of the containing class (an object). You cannot access it from the class directly (without first creating an instance, in other words).

Listing 5.6 shows an updated look at the `Program` class and its utilization of the `Employee` class, and Output 5.1 shows the results.

LISTING 5.6: Accessing Fields

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      Employee employee2;
      employee2 = new Employee();

      employee1.FirstName = "Inigo";
      employee1.LastName = "Montoya";
      employee1.Salary = "Too Little";
      IncreaseSalary(employee1);
      Console.WriteLine(
          "{0} {1}: {2}",
          employee1.FirstName,
          employee1.LastName,
          employee1.Salary);
      // ...
  }

  static void IncreaseSalary(Employee employee)
  {
      employee.Salary = "Enough to survive on";
  }
}
```

OUTPUT 5.1

```
Inigo Montoya: Enough to survive on
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Listing 5.6 instantiates two `Employee` objects, as you saw before. Next, it sets each field, calls `IncreaseSalary()` to change the salary, and then displays each field associated with the object referenced by `employee1`.

Notice that you first have to specify which `Employee` instance you are working with. Therefore, the `employee1` variable appears as a prefix to the field name when assigning and accessing the field.

Instance Methods

One alternative to formatting the names in the `WriteLine()` method call within `Main()` is to provide a method in the `Employee` class that takes care of the formatting. Changing the functionality to be within the `Employee` class rather than a member of `Program` is consistent with the encapsulation of a class. Why not group the methods relating to the employee's full name with the class that contains the data that forms the name?

Listing 5.7 demonstrates the creation of such a method.

LISTING 5.7: Accessing Fields from within the Containing Class

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;

  public string GetName()
  {
      return $"{ FirstName }  { LastName }";
  }
}
```

There is nothing particularly special about this method compared to what you learned in Chapter 4, except that now the `GetName()` method accesses fields on the object instead of just local variables. In addition, the method declaration is not marked with `static`. As you will see later in this chapter, static methods cannot directly access instance fields within a class. Instead, it is necessary to obtain an instance of the class to call any instance member, whether a method or a field.

Given the addition of the `GetName()` method, you can update `Program.Main()` to use the method, as shown in Listing 5.8 and Output 5.2.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Listing 5.8: Accessing Fields from outside the Containing Class

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      Employee employee2;
      employee2 = new Employee();

      employee1.FirstName = "Inigo";
      employee1.LastName = "Montoya";
      employee1.Salary = "Too Little";
      IncreaseSalary(employee1);
      Console.WriteLine(
          $"{ employee1.GetName() }: { employee1.Salary }");
      // ...
  }
  // ...
}
```

Output 5.2

```
Inigo Montoya: Enough to survive on
```

Using the `this` Keyword

You can obtain the reference to a class from within instance members that belong to the class. To indicate explicitly that the field or method accessed is an instance member of the containing class in C#, you use the keyword `this`. Use of `this` is implicit when calling any instance member, and it returns an instance of the object itself.

For example, consider the `SetName()` method shown in Listing 5.9.

Listing 5.9: Using `this` to Identify the Field's Owner Explicitly

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;

  public string GetName()
  {
      return $"{ FirstName }  { LastName }";
  }

  public void SetName(
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
    }
}
```

This example uses the keyword `this` to indicate that the fields `FirstName` and `LastName` are instance members of the class.

Although the `this` keyword can prefix any and all references to local class members, the general guideline is not to clutter code when there is no additional value. Therefore, you should avoid using the `this` keyword unless it is required. Listing 5.12 (later in this chapter) is an example of one of the few circumstances when such a requirement exists. Listings 5.9 and 5.10, however, are not good examples. In Listing 5.9, `this` can be dropped entirely without changing the meaning of the code. And in Listing 5.10 (presented next), by changing the naming convention for fields, we can avoid any ambiguity between local variables and fields.

BEGINNER TOPIC

Relying on Coding Style to Avoid Ambiguity

In the `SetName()` method, you did not have to use the `this` keyword because `FirstName` is obviously different from `newFirstName`. But suppose that, instead of calling the parameter "newFirstName," you called it "FirstName" (using PascalCase), as shown in Listing 5.10.

LISTING 5.10: Using `this` to Avoid Ambiguity

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;

  public string GetName()
  {
      return $"{ FirstName }  { LastName }";
  }

  // Caution:  Parameter names use PascalCase
  public void SetName(string FirstName, string LastName)
  {
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        this.FirstName = FirstName;
        this.LastName = LastName;
    }
}
```

In this example, it is not possible to refer to the `FirstName` field without explicitly indicating that the `Employee` object owns the variable. `this` acts just like the `employee1` variable prefix used in the `Program.Main()` method (see Listing 5.8); it identifies the reference as the one on which `SetName()` was called.

Listing 5.10 does not follow the C# naming convention in which parameters are declared like local variables, using camelCase. This can lead to subtle bugs, because assigning `FirstName` (intending to refer to the field) to `FirstName` (the parameter) will lead to code that still compiles and even runs. To avoid this problem, it is a good practice to have a different naming convention for parameters and local variables than the naming convention for fields. We demonstrate one such convention later in this chapter.

Language Contrast: Visual Basic—Accessing a Class Instance with `Me`

The C# keyword `this` is identical to the Visual Basic keyword `Me`.

In Listing 5.9 and Listing 5.10, the `this` keyword is not used in the `GetName()` method—it is optional. However, if local variables or parameters exist with the same name as the field (see the `SetName()` method in Listing 5.10), omitting `this` would result in accessing the local variable/parameter when the intention was the field; given this scenario, use of `this` is required.

You also can use the keyword `this` to access a class's methods explicitly. For example, `this.GetName()` is allowed within the `SetName()` method, permitting you to print out the newly assigned name (see Listing 5.11 and Output 5.3).

LISTING 5.11: Using `this` with a Method

```
class Employee
{
  // ...
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    public string GetName()
    {
        return $"{ FirstName }  { LastName }";
    }

    public void SetName(string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
        Console.WriteLine(
            $"Name changed to '{ this.GetName() }'");
    }
}
```

```
class Program
{
    static void Main()
    {
        Employee employee = new Employee();

        employee.SetName("Inigo", "Montoya");
        // ...
    }
    // ...
}
```

OUTPUT 5.3

```
Name changed to 'Inigo Montoya'
```

Sometimes it may be necessary to use `this` to pass a reference to the currently executing object. Consider the `Save()` method in Listing 5.12.

LISTING 5.12: Passing this in a Method Call

```
class Employee
{
    public string FirstName;
    public string LastName;
    public string Salary;

    public void Save()
    {
        DataStorage.Store(this);
    }
}
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
class DataStorage
{
  // Save an employee object to a file
  // named with the Employee name.
  public static void Store(Employee employee)
  {
      // ...
  }
}
```

The `Save()` method in Listing 5.12 calls a method on the `DataStorage` class, called `Store()`. The `Store()` method, however, needs to be passed the `Employee` object, which needs to be persisted. This is done using the keyword `this`, which passes the instance of the `Employee` object on which `Save()` was called.

ADVANCED TOPIC

Storing and Loading with Files

The actual implementation of the `Store()` method inside `DataStorage` involves classes within the `System.IO` namespace, as shown in Listing 5.13. Inside `Store()`, you begin by instantiating a `FileStream` object that you associate with a file corresponding to the employee's full name. The `FileMode.Create` parameter indicates that you want a new file to be created if there isn't already one with the `<firstname><lastname>.dat` name; if the file exists already, it will be overwritten. Next, you create a `StreamWriter` class. The `StreamWriter` class is responsible for writing text into the `FileStream`. You write the data using `WriteLine()` methods, just as though writing to the console.

LISTING 5.13: Data Persistence to a File

```
using System;
// IO namespace
using System.IO;

class DataStorage
{
  // Save an employee object to a file
  // named with the Employee name.
  // Error handling not shown.
  public static void Store(Employee employee)
  {
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        // Instantiate a FileStream using FirstNameLastName.dat
        // for the filename. FileMode.Create will force
        // a new file to be created or override an
        // existing file.
        FileStream stream = new FileStream(
            employee.FirstName + employee.LastName + ".dat",
            FileMode.Create);[1]

        // Create a StreamWriter object for writing text
        // into the FileStream.
        StreamWriter writer = new StreamWriter(stream);

        // Write all the data associated with the employee.
        writer.WriteLine(employee.FirstName);
        writer.WriteLine(employee.LastName);
        writer.WriteLine(employee.Salary);

        // Close the StreamWriter and its stream.
        writer.Close();  // Automatically closes the stream
    }
    // ...
}
```

Once the write operations are completed, both the `FileStream` and the `StreamWriter` need to be closed so that they are not left open indefinitely while waiting for the garbage collector to run. Listing 5.13 does not include any error handling, so if an exception is thrown, neither `Close()` method will be called.

The load process is similar (see Listing 5.14).

LISTING 5.14: Data Retrieval from a File

```
class Employee
{
  // ...
}
```

```
// IO namespace
using System;
using System.IO;

class DataStorage
{
  // ...

  public static Employee Load(string firstName, string lastName)
```

1. This code could be improved with a `using` statement, a construct that we have avoided because it has not yet been introduced.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    {
        Employee employee = new Employee();

        // Instantiate a FileStream using FirstNameLastName.dat
        // for the filename. FileMode.Open will open
        // an existing file or else report an error.
        FileStream stream = new FileStream(
            firstName + lastName + ".dat", FileMode.Open);[2]

        // Create a SteamReader for reading text from the file.
        StreamReader reader = new StreamReader(stream);

        // Read each line from the file and place it into
        // the associated property.
        employee.FirstName = reader.ReadLine();
        employee.LastName = reader.ReadLine();
        employee.Salary = reader.ReadLine();

        // Close the StreamReader and its stream.
        reader.Close();  // Automatically closes the stream

        return employee;
    }
}
```

```
class Program
{
  static void Main()
  {
        Employee employee1;

        Employee employee2 = new Employee();
        employee2.SetName("Inigo", "Montoya");
        employee2.Save();

       // Modify employee2 after saving.
        IncreaseSalary(employee2);

        // Load employee1 from the saved version of employee2
        employee1 = DataStorage.Load("Inigo", "Montoya");

        Console.WriteLine(
            $"{ employee1.GetName() }: { employee1.Salary }");

        // ...
  }
  // ...
}
```

2. This code could be improved with a `using` statement, a construct that we have avoided because it has not yet been introduced.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Output 5.4 shows the results.

Output 5.4

```
Name changed to 'Inigo Montoya'
Inigo Montoya:
```

The reverse of the save process appears in Listing 5.14, which uses a `StreamReader` rather than a `StreamWriter`. Again, `Close()` needs to be called on both `FileStream` and `StreamReader` once the data has been read.

Output 5.4 does not show any salary after `Inigo Montoya:` because `Salary` was not set to `Enough to survive on` by a call to `IncreaseSalary()` until after the call to `Save()`.

Notice in `Main()` that we can call `Save()` from an instance of an employee, but to load a new employee we call `DataStorage.Load()`. To load an employee, we generally don't already have an employee instance to load into, so an instance method on `Employee` would be less than ideal. An alternative to calling `Load` on `DataStorage` would be to add a static `Load()` method (see the section "Static Members," later in this chapter) to `Employee` so that it would be possible to call `Employee.Load()` (using the `Employee` class, not an instance of `Employee`).

Notice the inclusion of the `using System.IO` directive at the top of the listing. This makes each `IO` class accessible without prefixing it with the full namespace.

Access Modifiers

When declaring a field earlier in the chapter, you prefixed the field declaration with the keyword `public`. `public` is an **access modifier** that identifies the level of encapsulation associated with the member it decorates. Five access modifiers are available: `public`, `private`, `protected`, `internal`, and `protected internal`. This section considers the first two.

BEGINNER TOPIC

Encapsulation Part 2: Information Hiding

Besides wrapping data and methods together into a single unit, encapsulation deals with hiding the internal details of an object's data and behavior.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

To some degree, methods do this; from outside a method, all that is visible to a caller is the method declaration. None of the internal implementation is visible. Object-oriented programming enables this further, however, by providing facilities for controlling the extent to which members are visible from outside the class. Members that are not visible outside the class are **private members.**

In object-oriented programming, *encapsulation* is the term for not only grouping data and behavior, but also hiding data and behavior implementation details within a class (the capsule) so that the inner workings of a class are not exposed. This reduces the chance that callers will modify the data inappropriately or program according to the implementation, only to have it change in the future.

The purpose of an access modifier is to provide encapsulation. By using `public`, you explicitly indicate that it is acceptable that the modified fields are accessible from outside the `Employee` class—in other words, that they are accessible from the `Program` class, for example.

Consider an `Employee` class that includes a `Password` field, however. It should be possible to call an `Employee` object and verify the password using a `Logon()` method. Conversely, it should not be possible to access the `Password` field on an `Employee` object from outside the class.

To define a `Password` field as hidden and inaccessible from outside the containing class, you use the keyword `private` for the access modifier, in place of `public` (see Listing 5.15). As a result, the `Password` field is not accessible from inside the `Program` class, for example.

LISTING 5.15: Using the `private` Access Modifier

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;
  private string Password;
  private bool IsAuthenticated;

  public bool Logon(string password)
  {
      if(Password == password)
      {
          IsAuthenticated = true;
      }
      return IsAuthenticated;
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    }

    public bool GetIsAuthenticated()
    {
        return IsAuthenticated;
    }
    // ...
}
```

```
class Program
{
    static void Main()
    {
        Employee employee = new Employee();

        employee.FirstName = "Inigo";
        employee.LastName = "Montoya";

        // ...

        // Password is private, so it cannot be
        // accessed from outside the class.
        // Console.WriteLine(
        //    ("Password = {0}", employee.Password);
    }
    // ...
}
```

Although this option is not shown in Listing 5.15, it is possible to decorate a method with an access modifier of `private` as well.

If no access modifier is placed on a class member, the declaration defaults to `private`. In other words, members are private by default and programmers need to specify explicitly that a member is to be public.

Properties

The preceding section, "Access Modifiers," demonstrated how you can use the `private` keyword to encapsulate a password, preventing access from outside the class. This type of encapsulation is often too strict, however. For example, sometimes you might need to define fields that external classes can only read, but whose values you can change internally. Alternatively, perhaps you want to allow access to write some data in a class, but you need to be able to validate changes made to the data. In yet another scenario, perhaps you need to construct the data on the fly. Traditionally, languages enabled the features found in these examples by marking fields as private and then providing getter and setter methods for accessing and

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

modifying the data. The code in Listing 5.16 changes both `FirstName` and `LastName` to private fields. Public getter and setter methods for each field allow their values to be accessed and changed.

LISTING 5.16: Declaring Getter and Setter Methods

```
class Employee
{

  private string FirstName;
  // FirstName getter
  public string GetFirstName()
  {
      return FirstName;
  }
  // FirstName setter
  public void SetFirstName(string newFirstName)
  {
      if(newFirstName != null && newFirstName != "")
      {
          FirstName = newFirstName;
      }
  }

  private string LastName;
  // LastName getter
  public string GetLastName()
  {
      return LastName;
  }
  // LastName setter
  public void SetLastName(string newLastName)
  {
      if(newLastName != null && newLastName != "")
      {
          LastName = newLastName;
      }
  }
  // ...
}
```

Unfortunately, this change affects the programmability of the `Employee` class. No longer can you use the assignment operator to set data within the class, nor can you access the data without calling a method.

Declaring a Property

Recognizing the frequency of this type of pattern, the C# designers provided explicit syntax for it. This syntax is called a **property** (see Listing 5.17 and Output 5.5).

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

LISTING 5.17: Defining Properties

```
class Program
{
  static void Main()
  {
      Employee employee = new Employee();

      // Call the FirstName property's setter.
      employee.FirstName = "Inigo";

      // Call the FirstName property's getter.
      System.Console.WriteLine(employee.FirstName);
  }
}
```

```
class Employee
{
  // FirstName property
  public string FirstName
  {
      get
      {
          return _FirstName;
      }
      set
      {
          _FirstName = value;
      }
  }
  private string _FirstName;

  // LastName property
  public string LastName
  {
      get
      {
          return _LastName;
      }
      set
      {
          _LastName = value;
      }
  }
  private string _LastName;
  // ...
}
```

OUTPUT 5.5

```
Inigo
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

The first thing to notice in Listing 5.17 is not the property code itself, but rather the code within the `Program` class. Although you no longer have the fields with the `FirstName` and `LastName` identifiers, you cannot see this by looking at the `Program` class. The API for accessing an employee's first and last names has not changed at all. It is still possible to assign the parts of the name using a simple assignment operator, for example (`employee.FirstName = "Inigo"`).

The key feature is that properties provide an API that looks programmatically like a field. In actuality, no such fields exist. A property declaration looks exactly like a field declaration, but following it are curly braces in which to place the property implementation. Two optional parts make up the property implementation. The `get` part defines the getter portion of the property. It corresponds directly to the `GetFirstName()` and `GetLastName()` functions defined in Listing 5.16. To access the `FirstName` property, you call `employee.FirstName`. Similarly, setters (the `set` portion of the implementation) enable the calling syntax of the field assignment:

```
employee.FirstName = "Inigo";
```

Property definition syntax uses three contextual keywords. You use the `get` and `set` keywords to identify either the retrieval or the assignment portion of the property, respectively. In addition, the setter uses the `value` keyword to refer to the right side of the assignment operation. When `Program.Main()` calls `employee.FirstName = "Inigo"`, therefore, `value` is set to `"Inigo"` inside the setter and can be used to assign `_FirstName`. Listing 5.17's property implementations are the most commonly used. When the getter is called (such as in `Console.WriteLine(employee.FirstName)`), the value from the field (`_FirstName`) is obtained and written to the console.

Begin 3.0

Automatically Implemented Properties

In C# 3.0, property syntax includes a shorthand version. Since a property with a single backing field that is assigned and retrieved by the get and set accessors is so trivial and common (see the implementations of `FirstName` and `LastName`), the C# 3.0 compiler (and higher) allows the declaration of a property without any accessor implementation or backing field declaration. Listing 5.18 demonstrates the syntax with the `Title` and `Manager` properties, and Output 5.6 shows the results.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

LISTING 5.18: Automatically Implemented Properties

```
class Program
{
  static void Main()
  {
        Employee employee1 =
            new Employee();
        Employee employee2 =
            new Employee();

        // Call the FirstName property's setter.
        employee1.FirstName = "Inigo";

        // Call the FirstName property's getter.
        System.Console.WriteLine(employee1.FirstName);

        // Assign an auto-implemented property
        employee2.Title = "Computer Nerd";
        employee1.Manager = employee2;

        // Print employee1's manager's title.
        System.Console.WriteLine(employee1.Manager.Title);
  }
}
```

```
class Employee
{
  // FirstName property
  public string FirstName
  {
      get
      {
          return _FirstName;
      }
      set
      {
          _FirstName = value;
      }
  }
  private string _FirstName;

  // LastName property
  public string LastName
  {
      get
      {
          return _LastName;
      }
      set
      {
          _LastName = value;
      }
```


This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    }
    private string _LastName;

    public string Title { get; set; }

    public Employee Manager { get; set; }

    public string Salary { get; set; } = "Not Enough";
    // ...
}
```

OUTPUT 5.6

```
Inigo
Computer Nerd
```

Auto-implemented properties provide for a simpler way of writing properties in addition to reading them. Furthermore, when it comes time to add something such as validation to the setter, any existing code that calls the property will not have to change, even though the property declaration will have changed to include an implementation.

End 3.0

Throughout the remainder of the book, we will frequently use this C# 3.0 or later syntax without indicating that it is a feature introduced in C# 3.0.

Begin 6.0

One final thing to note about automatically declared properties is that in C# 6.0, it is possible to initialize them as Listing 5.18 does for `Salary`:

```
public string Salary { get; set; } = "Not Enough";
```

Prior to C# 6.0, property initialization was possible only via a method (including the constructor, as we discuss later in the chapter). However, with C# 6.0, you can initialize automatically implemented properties at declaration time using a syntax much like that used for field initialization.

End 6.0

Property and Field Guidelines

Given that it is possible to write explicit setter and getter methods rather than properties, on occasion a question may arise as to whether it is better to use a property or a method. The general guideline is that methods should represent actions and properties should represent data. Properties are intended to provide simple access to simple data with a simple

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

computation. The expectation is that invoking a property will not be significantly more expensive than accessing a field.

With regard to naming, notice that in Listing 5.18 the property name is `FirstName`, and the field name changed from earlier listings to `_FirstName`—that is, PascalCase with an underscore suffix. Other common naming conventions for the private field that backs a property are `_firstName` and `m_FirstName` (a holdover from C++, where the *m* stands for member variable), and on occasion the camelCase convention, just like with local variables.[3] The camelCase convention should be avoided, however. The camelCase used for property names is the same as the naming convention used for local variables and parameters, meaning that overlaps in names become highly probable. Also, to respect the principles of encapsulation, fields should not be declared as public or protected.

Guidelines

DO use properties for simple access to simple data with simple computations.

AVOID throwing exceptions from property getters.

DO preserve the original property value if the property throws an exception.

DO favor automatically implemented properties over properties with simple backing fields when no additional logic is required.

Regardless of which naming pattern you use for private fields, the coding standard for properties is PascalCase. Therefore, properties should use the `LastName` and `FirstName` pattern with names that represent nouns, noun phrases, or adjectives. It is not uncommon, in fact, that the property name is the same as the type name. Consider an `Address` property of type `Address` on a `Person` object, for example.

---

3. We prefer `_FirstName` because the *m* in front of the name is unnecessary when compared with an underscore (_). Also, by using the same casing as the property, it is possible to have only one string within the Visual Studio code template expansion tools, instead of having one for both the property name and the field name.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Guidelines

CONSIDER using the same casing on a property's backing field as that used in the property, distinguishing the backing field with an "_" prefix. Do not, however, use two underscores; identifiers beginning with two underscores are reserved for the use of the C# compiler itself.

DO name properties using a noun, noun phrase, or adjective.

CONSIDER giving a property the same name as its type.

AVOID naming fields with camelCase.

DO favor prefixing Boolean properties with "Is," "Can," or "Has," when that practice adds value.

DO NOT declare instance fields that are public or protected. (Instead, expose them via a property.)

DO name properties with PascalCase.

DO favor automatically implemented properties over fields.

DO favor automatically implemented properties over using fully expanded ones if there is no additional implementation logic.

Using Properties with Validation

Notice in Listing 5.19 that the `Initialize()` method of `Employee` uses the property rather than the field for assignment as well. Although this is not required, the result is that any validation within the property setter will be invoked both inside and outside the class. Consider, for example, what would happen if you changed the `LastName` property so that it checked `value` for `null` or an empty string, before assigning it to `_LastName`.

LISTING 5.19: Providing Property Validation

```
class Employee
{
  // ...
  public void Initialize(
      string newFirstName, string newLastName)
  {
      // Use property inside the Employee
      // class as well.
      FirstName = newFirstName;
      LastName = newLastName;
  }


  // LastName property
  public string LastName
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    {
        get
        {
            return _LastName;
        }
        set
        {
            // Validate LastName assignment
            if(value == null)
            {
                // Report error
                // In C# 6.0 replace "value" with nameof(value)
                throw new ArgumentNullException("value");
            }
            else
            {
                // Remove any whitespace around
                // the new last name.
                value = value.Trim();
                if(value == "")
                {
                    // Report error
                    // In C# 6.0 replace "value" with nameof(value)
                    throw new ArgumentException(
                        "LastName cannot be blank.", "value");[4]
                }
                else
                    _LastName = value;
            }
        }
    }
    private string _LastName;
    // ...
}
```

With this new implementation, the code throws an exception if `LastName` is assigned an invalid value, either from another member of the same class or via a direct assignment to `LastName` from inside `Program.Main()`. The ability to intercept an assignment and validate the parameters by providing a field-like API is one of the advantages of properties.

It is a good practice to access a property-backing field only from inside the property implementation. In other words, you should always use the property, rather than calling the field directly. In many cases, this principle holds even from code within the same class as the property. If you follow

4. Apologies to Teller, Cher, Sting, Madonna, Bono, Prince, Liberace, et al.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

this practice, when you add code such as validation code, the entire class immediately takes advantage of it.[5]

Although rare, it is possible to assign `value` inside the setter, as Listing 5.19 does. In this case, the call to `value.Trim()` removes any whitespace surrounding the new last name value.

Guidelines

AVOID accessing the backing field of a property outside the property, even from within the containing class.

DO use "value" for the `paramName` argument when calling the `ArgumentException()` or `ArgumentNullException()` constructor ("value" is the implicit name of the parameter on property setters).

Begin 6.0

ADVANCED TOPIC

nameof Operator

If during property validation you determine that the new value assignment is invalid, it is necessary to throw an exception—generally of type `ArgumentException()` or `ArgumentNullException()`. Both of these exceptions take an argument of type `string` called `paramName` that identifies the name of the parameter that is invalid. In Listing 5.19, we pass `"value"` as the argument for this parameter, but C# 6.0 provides an improvement with the `nameof` operator. The `nameof` operator takes an identifier, like the `value` variable, and returns a string representation of that name—in this case, `"value"`.

The advantage of using the `nameof` operator is that if the identifier name changes, then refactoring tools will automatically change the argument to `nameof` as well. If no refactoring tool is used, the code will no longer compile, forcing the developer to change the argument manually.

In the case of a property validator, the "parameter" is always `value` and cannot be changed, so the benefits of leveraging the `nameof` operator are

5. As described later in the chapter, one exception to this occurs when the field is marked as read-only, because then the value can be set only in the constructor. In C# 6.0, you can directly assign the value of a read-only property, completely eliminating the need for the read-only field.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

arguably lost. Nonetheless, consider continued use of the nameof operator in all cases of the paramName argument to remain consistent with the guideline: Always use nameof for the paramName argument passed into exceptions like ArgumentNullException and ArgumentNullException that take such a parameter.

End 6.0

Read-Only and Write-Only Properties

By removing either the getter or the setter portion of a property, you can change a property's accessibility. Properties with only a setter are write-only, which is a relatively rare occurrence. Similarly, providing only a getter will cause the property to be read-only; any attempts to assign a value will cause a compile error. To make Id read-only, for example, you would code it as shown in Listing 5.20.

LISTING 5.20: Defining a Read-Only Property prior to C# 6.0

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      employee1.Initialize(42);

      // ERROR:  Property or indexer 'Employee.Id'
      // cannot be assigned to; it is read-only.
      // employee1.Id = "490";
  }
}

class Employee
{
  public void Initialize(int id)
  {
      // Use field because Id property has no setter;
      // it is read-only.
      _Id = id.ToString();
  }

  // ...
  // Id property declaration
  public string Id
  {
      get
      {
          return _Id;
      }
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        // No setter provided.
    }
    private string _Id;

}
```

Listing 5.20 assigns the field from within the `Employee` constructor rather than the property (`_Id = id`). Assigning via the property causes a compile error, as it does in `Program.Main()`.

Begin 6.0

Starting in C# 6.0, there is also support for read-only **automatically implemented properties** as follows:

```
public bool[,,] Cells { get; } = new bool[2, 3, 3];
```

This is clearly a significant improvement over the pre-C# 6.0 approach, especially given the commonality of read-only properties for something like an array of items or the `Id` in Listing 5.20.

One important note about a read-only automatically implemented property is that, like read-only fields, the compiler requires that it be initialized in the constructor or via an initializer. In the preceding snippet we use an initializer, but the assignment of `Cells` from within the constructor is also permitted.

Given the guideline that fields should not be accessed from outside their wrapping property, those programming in a C# 6.0 world will discover that there is virtually no need to ever use pre-C# 6.0 syntax; instead, the programmer can always use a read-only, automatically implemented property. The only exception might be when the data type of the read-only modified field does not match the data type of the property—for example, if the field was of type `int` and the read-only property was of type `double`.

Guidelines

DO create read-only properties if the property value should not be changed.

DO create read-only automatically implemented properties in C# 6.o (or later) rather than read-only properties with a backing field if the property value should not be changed

End 6.0

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Properties As Virtual Fields

As you have seen, properties behave like virtual fields. In some instances, you do not need a backing field at all. Instead, the property getter returns a calculated value while the setter parses the value and persists it to some other member fields (if it even exists). Consider, for example, the `Name` property implementation shown in Listing 5.21. Output 5.7 shows the results.

LISTING 5.21: Defining Properties

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();

      employee1.Name = "Inigo Montoya";
      System.Console.WriteLine(employee1.Name);

      // ...
  }
}
```

```
class Employee
{
  // ...

  // FirstName property
  public string FirstName
  {
      get
      {
          return _FirstName;
      }
      set
      {
          _FirstName = value;
      }
  }
  private string _FirstName;

  // LastName property
  public string LastName
  {
      get
      {
          return _LastName;
      }
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        set
        {
            _LastName = value;
        }
    }
    private string _LastName;
    // ...

    // Name property
    public string Name
    {
        get
        {
            return $"{ FirstName } { LastName }";
        }
        set
        {
            // Split the assigned value into
            // first and last names.
            string[] names;
            names = value.Split(new char[]{' '});
            if(names.Length == 2)
            {
                FirstName = names[0];
                LastName = names[1];
            }
            else
            {
                // Throw an exception if the full
                // name was not assigned.
               throw new System. ArgumentException (
                    $"Assigned value '{ value }' is invalid", "value");
            }
        }
    }
    public string Initials => $"{ FirstName[0] } { LastName[0] }";
    // ...
}
```

OUTPUT 5.7

```
Inigo Montoya
```

The getter for the `Name` property concatenates the values returned from the `FirstName` and `LastName` properties. In fact, the name value assigned is not actually stored. When the `Name` property is assigned, the value on the right side is parsed into its first and last name parts.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Begin 2.0

Access Modifiers on Getters and Setters

As previously mentioned, it is a good practice not to access fields from outside their properties because doing so circumvents any validation or additional logic that may be inserted. Unfortunately, C# 1.0 did not allow different levels of encapsulation between the getter and setter portions of a property. It was not possible, therefore, to create a public getter and a private setter so that external classes would have read-only access to the property while code within the class could write to the property.

In C# 2.0, support was added for placing an access modifier on either the get or the set portion of the property implementation (not on both), thereby overriding the access modifier specified on the property declaration. Listing 5.22 demonstrates how to do this.

LISTING 5.22: Placing Access Modifiers on the Setter

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      employee1.Initialize(42);
      // ERROR: The property or indexer 'Employee.Id'
      // cannot be used in this context because the set
      // accessor is inaccessible
      employee1.Id = "490";
  }
}
```

```
class Employee
{
  public void Initialize(int id)
  {
      // Set Id property
      Id = id.ToString();
  }

  // ...
  // Id property declaration
  public string Id
  {
      get
      {
          return _Id;
      }
      // Providing an access modifier is possible in C# 2.0
      // and higher only
      private set
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        {
            _Id = value;
        }
    }
    private string _Id;

}
```

By using `private` on the setter, the property appears as read-only to classes other than `Employee`. From within `Employee`, the property appears as read/write, so you can assign the property within the constructor. When specifying an access modifier on the getter or setter, take care that the access modifier is more restrictive than the access modifier on the property as a whole. It is a compile error, for example, to declare the property as `private` and the setter as `public`.

Guidelines

DO apply appropriate accessibility modifiers on implementations of getters and setters on all properties.

DO NOT provide set-only properties or properties with the setter having broader accessibility than the getter.

End 2.0

Properties and Method Calls Not Allowed As `ref` or `out` Parameter Values

C# allows properties to be used identically to fields, except when they are passed as `ref` or `out` parameter values. `ref` and `out` parameter values are internally implemented by passing the memory address to the target method. However, because properties can be virtual fields that have no backing field, or can be read-only or write-only, it is not possible to pass the address for the underlying storage. As a result, you cannot pass properties as `ref` or `out` parameter values. The same is true for method calls. Instead, when code needs to pass a property or method call as a `ref` or `out` parameter value, the code must first copy the value into a variable and then pass the variable. Once the method call has completed, the code must assign the variable back into the property.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

ADVANCED TOPIC

Property Internals

Listing 5.23 shows that getters and setters are exposed as `get_FirstName()` and `set_FirstName()` in the CIL.

LISTING 5.23: CIL Code Resulting from Properties

```
// ...

.field private string _FirstName
.method public hidebysig specialname instance string
        get_FirstName() cil managed
{
  // Code size       12 (0xc)
  .maxstack  1
  .locals init (string V_0)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldfld      string Employee::_FirstName
  IL_0007:  stloc.0
  IL_0008:  br.s       IL_000a

  IL_000a:  ldloc.0
  IL_000b:  ret
} // end of method Employee::get_FirstName

.method public hidebysig specialname instance void
        set_FirstName(string 'value') cil managed
{
  // Code size       9 (0x9)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldarg.1
  IL_0003:  stfld      string Employee::_FirstName
  IL_0008:  ret
} // end of method Employee::set_FirstName

.property instance string FirstName()
{
  .get instance string Employee::get_FirstName()
  .set instance void Employee::set_FirstName(string)
} // end of property Employee::FirstName

// ...
```

Just as important to their appearance as regular methods is the fact that properties are an explicit construct within the CIL, too. As Listing 5.24

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

shows, the getters and setters are called by CIL properties, which are an explicit construct within the CIL code. Because of this, languages and compilers are not restricted to always interpreting properties based on a naming convention. Instead, CIL properties provide a means for compilers and code editors to provide special syntax.

LISTING 5.24: Properties Are an Explicit Construct in CIL

```
.property instance string FirstName()
{
  .get instance string Program::get_FirstName()
  .set instance void Program::set_FirstName(string)
} // end of property Program::FirstName
```

Notice in Listing 5.23 that the getters and setters that are part of the property include the `specialname` metadata. This modifier is what IDEs, such as Visual Studio, use as a flag to hide the members from IntelliSense.

Begin 3.0

An automatically implemented property is almost identical to one for which you define the backing field explicitly. In place of the manually defined backing field, the C# compiler generates a field with the name `<PropertyName>k_BackingField` in IL. This generated field includes an attribute (see Chapter 17) called `System.Runtime.CompilerServices.CompilerGeneratedAttribute`. Both the getters and the setters are decorated with the same attribute because they, too, are generated—with the same implementation as in Listings 5.23 and 5.24.

End 3.0

Constructors

Now that you have added fields to a class and can store data, you need to consider the validity of that data. As you saw in Listing 5.3, it is possible to instantiate an object using the `new` operator. The result, however, is the ability to create an employee with invalid data. Immediately following the assignment of `employee`, you have an `Employee` object whose name and salary are not initialized. In this particular listing, you assigned the uninitialized fields immediately following the instantiation of an employee, but if you failed to do the initialization, you would not receive a warning from the compiler. As a result, you could end up with an `Employee` object with an invalid name.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Declaring a Constructor

To correct this problem, you need to provide a means of specifying the required data when the object is created. You do this using a constructor as demonstrated in Listing 5.25.

Listing 5.25: Defining a Constructor

```
class Employee
{
  // Employee constructor
  public Employee(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }

  public string FirstName{ get; set; }
  public string LastName{ get; set; }
  public string Salary{ get; set; } = "Not Enough";

  // ...
}
```

As shown here, to define a constructor you create a method with no return type, whose method name is identical to the class name.

The constructor is the method that the runtime calls to initialize an instance of the object. In this case, the constructor takes the first name and the last name as parameters, allowing the programmer to specify these names when instantiating the `Employee` object. Listing 5.26 is an example of how to call a constructor.

Listing 5.26: Calling a Constructor

```
class Program
{
  static void Main()
  {
      Employee employee;
      employee = new Employee("Inigo", "Montoya");
      employee.Salary = "Too Little";

      System.Console.WriteLine(
          "{0} {1}: {2}",
          employee.FirstName,
          employee.LastName,
          employee.Salary);
  }
  // ...
}
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Notice that the `new` operator returns the type of the object being instantiated (even though no return type or return statement was specified explicitly in the constructor's declaration or implementation). In addition, you have removed the initialization code for the first and last names because that initialization takes place within the constructor. In this example, you don't initialize `Salary` within the constructor, so the code assigning the salary still appears.

Developers should take care when using both assignment at declaration time and assignment within constructors. Assignments within the constructor will occur after any assignments are made when a field is declared (such as `string Salary = "Not enough"` in Listing 5.5). Therefore, assignment within a constructor will override any value assigned at declaration time. This subtlety can lead to a misinterpretation of the code by a casual reader who assumes the value after instantiation is the one assigned in the field declaration. Therefore, it is worth considering a coding style that does not mix both declaration assignment and constructor assignment within the same class.

ADVANCED TOPIC

Implementation Details of the new Operator

Internally, the interaction between the `new` operator and the constructor is as follows. The `new` operator retrieves "empty" memory from the memory manager and then calls the specified constructor, passing a reference to the empty memory to the constructor as the implicit `this` parameter. Next, the remainder of the constructor chain executes, passing around the reference between constructors. None of the constructors have a return type; behaviorally they all return `void`. When execution completes on the constructor chain is complete, the `new` operator returns the memory reference, now referring to the memory in its initialized form.

Default Constructors

When you add a constructor explicitly, you can no longer instantiate an `Employee` from within `Main()` without specifying the first and last names. The code shown in Listing 5.27, therefore, will not compile.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Listing 5.27: Default Constructor No Longer Available

```
class Program
{
  static void Main()
  {
      Employee employee;
      // ERROR: No overload because method 'Employee'
      // takes '0' arguments.
      employee = new Employee();

      // ...
  }
}
```

If a class has no explicitly defined constructor, the C# compiler adds one during compilation. This constructor takes no parameters and, therefore, is the **default constructor** by definition. As soon as you add an explicit constructor to a class, the C# compiler no longer provides a default constructor. Therefore, with `Employee(string firstName, string lastName)` defined, the default constructor, `Employee()`, is not added by the compiler. You could manually add such a constructor, but then you would again be allowing construction of an `Employee` without specifying the employee name.

It is not necessary to rely on the default constructor defined by the compiler. It is also possible for programmers to define a default constructor explicitly—perhaps one that initializes some fields to particular values. Defining the default constructor simply involves declaring a constructor that takes no parameters.

Object Initializers

Begin 3.0

Starting with C# 3.0, the C# language team added functionality to initialize an object's accessible fields and properties using an **object initializer.** The object initializer consists of a set of member initializers enclosed in curly braces following the constructor call to create the object. Each member initializer is the assignment of an accessible field or property name with a value (see Listing 5.28).

Listing 5.28: Calling an Object Initializer

```
class Program
{
  static void Main()
  {
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
            Employee employee1 = new Employee("Inigo", "Montoya")
                { Title = "Computer Nerd", Salary = "Not enough"};
            // ...
        }
    }
```

Notice that the same constructor rules apply even when using an object initializer. In fact, the resultant CIL is exactly the same as it would be if the fields or properties were assigned within separate statements immediately following the constructor call. The order of member initializers in C# provides the sequence for property and field assignment in the statements following the constructor call within CIL.

In general, all properties should be initialized to reasonable default values by the time the constructor exits. Moreover, by using validation logic on the setter, it is possible to restrict the assignment of invalid data to a property. On occasion, the values on one or more properties may cause other properties on the same object to contain invalid values. When this occurs, exceptions from the invalid state should be postponed until the invalid interrelated property values become relevant.

Guidelines

DO provide sensible defaults for all properties, ensuring that defaults do not result in a security hole or significantly inefficient code.
For automatically implemented properties, set the default via the constructor.

DO allow properties to be set in any order, even if this results in a temporarily invalid object state.

3.0

ADVANCED TOPIC

Collection Initializers

Using a similar syntax to that of object initializers, collection initializers were added in C# 3.0. Collection initializers support a similar feature set as object initializers, only with collections. Specifically, a collection initializer allows the assignment of items within the collection at the time of the collection's instantiation. Borrowing on the same syntax used for arrays, the collection initializer initializes each item within the collection as part

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

of collection creation. Initializing a list of `Employee`s, for example, involves specifying each item within curly braces following the constructor call, as Listing 5.29 shows.

LISTING 5.29: Calling an Object Initializer

```
class Program
{
  static void Main()
  {
      List<Employee> employees = new List<Employee>()
         {
            new Employee("Inigo", "Montoya"),
            new Employee("Kevin", "Bost")
         };
      // ...
  }
}
```

After the assignment of a new collection instance, the compiler-generated code instantiates each object in sequence and adds them to the collection via the `Add()` method.

End 3.0

ADVANCED TOPIC

Finalizers

Constructors define what happens during the instantiation process of a class. To define what happens when an object is destroyed, C# provides the finalizer construct. Unlike destructors in C++, finalizers do not run immediately after an object goes out of scope. Rather, the finalizer executes at some unspecified time after an object is determined to be "unreachable." Specifically, the garbage collector identifies objects with finalizers during a garbage collection cycle, and instead of immediately de-allocating those objects, it adds them to a finalization queue. A separate thread runs through each object in the finalization queue and calls the object's finalizer before removing it from the queue and making it available for the garbage collector again. Chapter 9 discusses this process, along with resource cleanup, in depth.

Overloading Constructors

Constructors can be overloaded—you can have more than one constructor as long as the number or types of the parameters vary. For example, as

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Listing 5.30 shows, you could provide a constructor that has an employee ID with first and last names, or even just the employee ID.

Listing 5.30: Overloading a Constructor

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      FirstName = firstName;
      LastName = lastName;
  }

  public Employee(
      int id, string firstName, string lastName )
  {
      Id = id;
      FirstName = firstName;
      LastName = lastName;
  }

  public Employee(int id)
  {
      Id = id;

      // Look up employee name...
      // ...
  }

  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Salary { get; set; } = "Not Enough";

  // ...
}
```

This approach enables `Program.Main()` to instantiate an employee from the first and last names either by passing in the employee ID only or by passing both the names and the IDs. You would use the constructor with both the names and the IDs when creating a new employee in the system. You would use the constructor with only the ID to load up the employee from a file or a database.

As is the case with method overloading, multiple constructors are used to support simple scenarios using a small number of parameters and complex scenarios with additional parameters. Consider using optional parameters in favor of overloading so that the default values for "defaulted"

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

properties are visible in the API. For example, a constructor signature of `Person(string firstName, string lastName, int? age = null)` provides signature documentation that if the `Age` of a `Person` is not specified, it will default to `null`.

Guidelines

DO use the same name for constructor parameters (camelCase) and properties (PascalCase) if the constructor parameters are used to simply set the property.

DO provide constructor optional parameters and/or convenience constructor overloads that initialize properties with good defaults.

DO allow properties to be set in any order, even if this results in a temporarily invalid object state.

Constructor Chaining: Calling Another Constructor Using `this`

Notice in Listing 5.30 that the initialization code for the `Employee` object is now duplicated in multiple places and, therefore, has to be maintained in multiple places. The amount of code is small, but there are ways to eliminate the duplication by calling one constructor from another—**constructor chaining**—using **constructor initializers.** Constructor initializers determine which constructor to call before executing the implementation of the current constructor (see Listing 5.31).

LISTING 5.31: Calling One Constructor from Another

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      FirstName = firstName;
      LastName = lastName;
  }

  public Employee(
      int id, string firstName, string lastName )
      : this(firstName, lastName)
  {
      Id = id;
  }

  public Employee(int id)
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    {
        Id = id;

        // Look up employee name...
        // ...

        // NOTE:  Member constructors cannot be
        // called explicitly inline
        // this(id, firstName, lastName);
    }

    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Salary { get; set; } = "Not Enough";

    // ...
}
```

To call one constructor from another within the same class (for the same object instance), C# uses a colon followed by the `this` keyword, followed by the parameter list on the callee constructor's declaration. In this case, the constructor that takes all three parameters calls the constructor that takes two parameters. Often, this calling pattern is reversed—that is, the constructor with the fewest parameters calls the constructor with the most parameters, passing defaults for the parameters that are not known.

BEGINNER TOPIC

Centralizing Initialization

Notice that in the `Employee(int id)` constructor implementation from Listing 5.31, you cannot call `this(firstName, LastName)` because no such parameters exist on this constructor. To enable such a pattern in which all initialization code happens through one method, you must create a separate method, as shown in Listing 5.32.

LISTING 5.32: Providing an Initialization Method

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      int id;
      // Generate an employee ID...
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        // ...
        Initialize(id, firstName, lastName);
    }

    public Employee(int id, string firstName, string lastName )
    {
        Initialize(id, firstName, lastName);
    }

    public Employee(int id)
    {
        string firstName;
        string lastName;
        Id = id;

        // Look up employee data
        // ...

        Initialize(id, firstName, lastName);
    }

    private void Initialize(
        int id, string firstName, string lastName)
    {
        Id = id;
        FirstName = firstName;
        LastName = lastName;
    }
    // ...
}
```

In this case, the method is called `Initialize()` and it takes both the names and the employee IDs. Note that you can continue to call one constructor from another, as shown in Listing 5.31.

ADVANCED TOPIC

Begin 3.0

Anonymous Types

C# 3.0 introduced support for anonymous types. These data types are generated by the compiler "on the fly," rather than through explicit class definitions. Listing 5.33 shows such a declaration.

LISTING 5.33: Implicit Local Variables with Anonymous Types

```
using System;

class Program
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
{
  static void Main()
  {
      var patent1 =
          new
          {
              Title = "Bifocals",
              YearOfPublication = "1784"
          };
      var patent2 =
          new
          {
              Title = "Phonograph",
              YearOfPublication = "1877"
          };
      var patent3 =
          new
          {
              patent1.Title,
              Year = patent1.YearOfPublication
          };

      System.Console.WriteLine("{0} ({1})",
          patent1.Title, patent1.YearOfPublication);
      System.Console.WriteLine("{0} ({1})",
          patent2.Title, patent1.YearOfPublication);

      Console.WriteLine();
      Console.WriteLine(patent1);
      Console.WriteLine(patent2);

      Console.WriteLine();
      Console.WriteLine(patent3);
  }
}
```

3.0

The corresponding output is shown in Output 5.8.

Output 5.8

```
Bifocals (1784)
Phonograph (1877)

{ Title = Bifocals, YearOfPublication = 1784 }
{ Title = Phonograph, YearOfPublication = 1877 }

{ Title = Bifocals, Year = 1784 }
```

Listing 5.33 demonstrates the assignment of an anonymous type to an implicitly typed (`var`) local variable. When the compiler encounters the

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

anonymous type syntax, it generates a class with properties corresponding to the named values and data types in the anonymous type declaration. Although there is no available name in C# for the generated type, it is still statically typed. For example, the properties of the type are fully accessible. In Listing 5.33, `patent1.Title` and `patent2.YearOfPublication` are called within the `Console.WriteLine()` statement. Any attempts to call nonexistent members will result in compile-time errors. Even IntelliSense in IDEs such as Visual Studio works with the anonymous type.

In Listing 5.33, member names on the anonymous types are explicitly identified using the assignment of the value to the name (see `Title` and `YearOfPublication` in the `patent1` and `patent2` assignments). However, if the value assigned is a property or field, the name will default to the name of the field or property if not specified explicitly. For example, `patent3` is defined using a property name "Title" rather than an assignment to an implicit name. As Output 5.8 shows, the resultant property name is determined by the compiler to match the property from where the value was retrieved.

Although the compiler allows anonymous type declarations such as the ones shown in Listing 5.33, you should generally avoid these kinds of declarations, and even the associated implicit typing with `var`, unless you are working with lambda and query expressions that associate data from different types or you are horizontally projecting the data so that for a particular type, there is less data overall. Until frequent querying of data held in collections makes explicit type declaration burdensome, it is preferable to explicitly declare types as outlined in this chapter.

End 3.0

Static Members

The `HelloWorld` example in Chapter 1 briefly touched on the keyword `static`. This section defines the `static` keyword more fully.

Let's consider an example. Assume that the employee `Id` value needs to be unique for each employee. One way to accomplish this is to store a counter to track each employee ID. If the value is stored as an instance field, however, every time you instantiate an object, a new `NextId` field will be created such that every instance of the `Employee` object will consume memory for that field. The biggest problem is that each time an `Employee` object is instantiated, the `NextId` value on all of the previously instantiated `Employee`

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

objects needs to be updated with the next ID value. In this case, what you need is a single field that all `Employee` object instances share.

Language Contrast: C++/Visual Basic—Global Variables and Functions

Unlike many of the languages that came before it, C# does not have global variables or global functions. All fields and methods in C# appear within the context of a class. The equivalent of a global field or function within the realm of C# is a static field or function. There is no functional difference between global variables/functions and C# static fields/methods, except that static fields/methods can include access modifiers, such as `private`, that can limit the access and provide better encapsulation.

Static Fields

To define data that is available across multiple instances, you use the `static` keyword, as demonstrated in Listing 5.34.

LISTING 5.34: Declaring a Static Field

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      FirstName = firstName;
      LastName = lastName;
      Id = NextId;
      NextId++;
  }

  // ...

  public static int NextId;
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Salary { get; set; } = "Not Enough";

  // ...
}
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

In this example, the NextId field declaration includes the static modifier and, therefore, is called a **static field.** Unlike Id, a single storage location for NextId is shared across all instances of Employee. Inside the Employee constructor, you assign the new Employee object's Id the value of NextId immediately before incrementing the Id. When another Employee class is created, NextId will be incremented and the new Employee object's Id field will hold a different value.

Just as **instance fields** (nonstatic fields) can be initialized at declaration time, so can static fields, as demonstrated in Listing 5.35.

LISTING 5.35: Assigning a Static Field at Declaration

```
class Employee
{
  // ...
  public static int NextId = 42;
  // ...
}
```

Unlike with instance fields, if no initialization for a static field is provided, the static field will automatically be assigned its default value (0, null, false, and so on)—the equivalent of default(T), where T is the name of the type. As a result, it will be possible to access the static field even if it has never been explicitly assigned in the C# code.

Nonstatic fields, or instance fields, provide a new storage location for each object to which they belong. In contrast, static fields don't belong to the instance, but rather to the class itself. As a result, you access a static field from outside a class via the class name. Consider the new Program class shown in Listing 5.36 (using the Employee class from Listing 5.34).

LISTING 5.36: Accessing a Static Field

```
using System;

class Program
{
  static void Main()
  {
      Employee.NextId = 1000000;

      Employee employee1 = new Employee(
          "Inigo", "Montoya");
      Employee employee2 = new Employee(
          "Princess", "Buttercup");
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        Console.WriteLine(
            "{0} {1} ({2})",
            employee1.FirstName,
            employee1.LastName,
            employee1.Id);
        Console.WriteLine(
            "{0} {1} ({2})",
            employee2.FirstName,
            employee2.LastName,
            employee2.Id);

        Console.WriteLine(
            $"NextId = { Employee.NextId }");
    }

    // ...
}
```

Output 5.9 shows the results of Listing 5.36.

Output 5.9

```
Inigo Montoya (1000000)
Princess Buttercup (1000001)
NextId = 1000002
```

To set and retrieve the initial value of the `NextId` static field, you use the class name, `Employee`, rather than a reference to an instance of the type. The only place you can omit the class name is within the class itself (or a derived class). In other words, the `Employee(...)` constructor did not need to use `Employee.NextId` because the code appeared within the context of the `Employee` class itself and, therefore, the context was already understood. The scope of a variable is the program text in which the variable can be referred to by its unqualified name; the scope of a static field is the text of the class (and any derived classes).

Even though you refer to static fields slightly differently than you refer to instance fields, it is not possible to define a static field and an instance field with the same name in the same class. The possibility of mistakenly referring to the wrong field is high, so the C# designers decided to prevent such code. Overlap in names, therefore, introduces conflict within the declaration space.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

BEGINNER TOPIC

Data Can Be Associated with Both a Class and an Object

Both classes and objects can have associated data, just as can the molds and the widgets created from them.

For example, a mold could have data corresponding to the number of widgets it created, the serial number of the next widget, the current color of the plastic injected into the mold, and the number of widgets it produces per hour. Similarly, a widget has its own serial number, its own color, and perhaps the date and time when the widget was created. Although the color of the widget corresponds to the color of the plastic within the mold at the time the widget was created, it obviously does not contain data corresponding to the color of the plastic currently in the mold, or the serial number of the next widget to be produced.

In designing objects, programmers should take care to declare both fields and methods appropriately, as static or instance based. In general, you should declare methods that don't access any instance data as static methods, and methods that access instance data (where the instance is not passed in as a parameter) as instance methods. Static fields store data corresponding to the class, such as defaults for new instances or the number of instances that have been created. Instance fields store data associated with the object.

Static Methods

Just like static fields, you access static methods directly off the class name—for example, as `Console.ReadLine()`. Furthermore, it is not necessary to have an instance to access the method.

Listing 5.37 provides another example of both declaring and calling a static method.

LISTING 5.37: Defining a Static Method on `DirectoryInfo`

```
public static class DirectoryInfoExtension
{
  public static void CopyTo(
      DirectoryInfo sourceDirectory, string target,
      SearchOption option, string searchPattern)
  {
      if (target[target.Length - 1] !=
            Path.DirectorySeparatorChar)
      {
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
            target += Path.DirectorySeparatorChar;
        }
        if (!Directory.Exists(target))
        {
            Directory.CreateDirectory(target);
        }

        for (int i = 0; i < searchPattern.Length; i++)
        {
            foreach (string file in
                Directory.GetFiles(
                    sourceDirectory.FullName, searchPattern))
            {
                File.Copy(file,
                    target + Path.GetFileName(file), true);
            }
        }

        //Copy subdirectories (recursively)
        if (option == SearchOption.AllDirectories)
        {
            foreach(string element in
                Directory.GetDirectories(
                    sourceDirectory.FullName))
            {
                Copy(element,
                    target + Path.GetFileName(element),
                    searchPattern);
            }
        }
    }
}
```

```
        // ...
        DirectoryInfo directory = new DirectoryInfo(".\\Source");
        directory.MoveTo(".\\Root");
        DirectoryInfoExtension.CopyTo(
            directory, ".\\Target",
            SearchOption.AllDirectories, "*");
        // ...
```

In Listing 5.37, the `DirectoryInfoExtension.Copy()` method takes a `DirectoryInfo` object and copies the underlying directory structure to a new location.

Because static methods are not referenced through a particular instance, the `this` keyword is invalid inside a static method. In addition, it is not possible to access either an instance field or an instance method directly from within a static method without a reference to the particular instance

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

to which the field or method belongs. (Note that `Main()` is another example of a static method.)

One might have expected this method on the `System.IO.Directory` class or as an instance method on `System.IO.DirectoryInfo`. Since neither exists, Listing 5.37 defines such a method on an entirely new class. In the section "Extension Methods" later in this chapter, we show how to make it appear as an instance method on `DirectoryInfo`.

Static Constructors

In addition to static fields and methods, C# supports **static constructors.** Static constructors are provided as a means to initialize the class itself, rather than the instances of a class. Such constructors are not called explicitly; instead, the runtime calls static constructors automatically upon first access to the class, whether by calling a regular constructor or by accessing a static method or field on the class. Because the static constructor cannot be called explicitly, no parameters are allowed on static constructors.

You use static constructors to initialize the static data within the class to a particular value, primarily when the initial value involves more complexity than a simple assignment at declaration time. Consider Listing 5.38.

LISTING 5.38: Declaring a Static Constructor

```
class Employee
{
  static Employee()
  {
      Random randomGenerator = new Random();
      NextId = randomGenerator.Next(101, 999);
  }

  // ...
  public static int NextId = 42;
  // ...
}
```

Listing 5.38 assigns the initial value of `NextId` to be a random integer between 100 and 1,000. Because the initial value involves a method call, the `NextId` initialization code appears within a static constructor and not as part of the declaration.

If assignment of `NextId` occurs within both the static constructor and the declaration, it is not obvious what the value will be when initialization

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

concludes. The C# compiler generates CIL in which the declaration assignment is moved to be the first statement within the static constructor. Therefore, `NextId` will contain the value returned by `randomGenerator.Next(101, 999)` instead of a value assigned during `NextId`'s declaration. Assignments within the static constructor, therefore, will take precedence over assignments that occur as part of the field declaration, as was the case with instance fields. Note that there is no support for defining a static finalizer.

Be careful not to throw an exception from a static constructor, as this will render the type unusable for the remainder of the application's lifetime.[6]

ADVANCED TOPIC

Favor Static Initialization during Declaration

Static constructors execute before the first access to any member of a class, whether it is a static field, another static member, or an instance constructor. To support this practice, the compiler injects a check into all type static members and constructors to ensure that the static constructor runs first.

Without the static constructor, the compiler initializes all static members to their default values and avoids adding the static constructor check. The result is that static assignment initialization is called before accessing any static fields but not necessarily before all static methods or any instance constructor is invoked. This might provide a performance improvement if initialization of static members is expensive and is not needed before accessing a static field. For this reason, you should consider initializing static fields inline rather than using a static constructor, or initializing them at declaration time.

Guidelines

CONSIDER initializing static fields inline rather than explicitly using static constructors or declaration assigned values.

6. Technically, the application domain's lifetime—the CLR's virtual equivalent of an operating system process.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Static Properties

Begin 2.0

You also can declare properties as static. For example, Listing 5.39 wraps the data for the next ID into a property.

LISTING 5.39: Declaring a Static Property

```
class Employee
{
  // ...
  public static int NextId
  {
      get
      {
          return _NextId;
      }
      private set
      {
          _NextId = value;
      }
  }
  public static int _NextId = 42;
  // ...
}
```

It is almost always better to use a static property rather than a public static field, because public static fields are callable from anywhere, whereas a static property offers at least some level of encapsulation.

In C# 6.0, the entire `NextId` implementation—including an inaccessible backing field—can be simplified down to an automatically implemented property with an initializer:

Begin 6.0

End 6.0

```
public static int NextId { get; private set; } = 42;
```

Static Classes

Some classes do not contain any instance fields. Consider, for example, a `Math` class that has functions corresponding to the mathematical operations `Max()` and `Min()`, as shown in Listing 5.40.

LISTING 5.40: Declaring a Static Class

```
// Static class introduced in C# 2.0
public static class SimpleMath
{
  // params allows the number of parameters to vary.
  public static int Max(params int[] numbers)
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    {
        // Check that there is at least one item in numbers.
        if(numbers.Length == 0)
        {
            throw new ArgumentException(
                "numbers cannot be empty", "numbers");
        }

        int result;
        result = numbers[0];
        foreach (int number in numbers)
        {
            if(number > result)
            {
                result = number;
            }
        }
        return result;
    }

    // params allows the number of parameters to vary.
    public static int Min(params int[] numbers)
    {
        // Check that there is at least one item in numbers.
        if(numbers.Length == 0)
        {
            throw new ArgumentException(
                "numbers cannot be empty", "numbers");
        }

        int result;
        result = numbers[0];
        foreach (int number in numbers)
        {
            if(number < result)
            {
                result = number;
            }
        }
        return result;
    }
}
```

2.0

```
public class Program
{
  public static void Main(string[] args)
  {
      int[] numbers = new int[args.Length];
      for (int count = 0; count < args.Length; count++)
      {
          numbers[count] = args[count].Length;
      }
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
        Console.WriteLine(
            $@"Longest argument length = {
                SimpleMath.Max(numbers) }");
        Console.WriteLine(
            $@"Shortest argument length = {
                SimpleMath.Min(numbers) }");
    }
}
```

This class does not have any instance fields (or methods), so creation of such a class would be pointless. Consequently, the class is decorated with the `static` keyword. The `static` keyword on a class provides two facilities. First, it prevents a programmer from writing code that instantiates the `SimpleMath` class. Second, it prevents the declaration of any instance fields or methods within the class. Because the class cannot be instantiated, instance members would be pointless. The `Program` class in prior listings is another good candidate for a static class because it too contains only static members.

End 2.0

One more distinguishing characteristic of the static class is that the C# compiler automatically marks it as `abstract` and `sealed` within the CIL. This designates the class as **inextensible;** in other words, no class can be derived from this class or even instantiate it.

In the previous chapter, we saw that the `using static` directive can be used with static classes such as `SimpleMath`. For example, adding a `using static SimpleMath;` declarative at the top of Listing 5.40 would allow you to invoke `Max` without the `SimpleMath` prefix:

Begin 6.0

Begin 3.0

```
Console.WriteLine(
    $@"Longest argument length = { Max(numbers) }");
```

End 6.0

Extension Methods

Consider the `System.IO.DirectoryInfo` class, which is used to manipulate filesystem directories. This class supports functionality to list the files and subdirectories (`DirectoryInfo.GetFiles()`) as well as the capability to move the directory (`DirectoryInfo.Move()`). One feature it doesn't support directly is the copy feature. If you needed such a method, you would have to implement it, as shown earlier in Listing 5.37.

The `DirectoryInfoExtension.Copy()` method is a standard static method declaration. However, notice that calling this `Copy()` method is different

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

from calling the `DirectoryInfo.Move()` method. This is unfortunate. Ideally, we want to add a method to `DirectoryInfo` so that, given an instance, we could call `Copy()` as an instance method—`directory.Copy()`.

C# 3.0 simulates the creation of an instance method on a different class via **extension methods.** To do this, we simply change the signature of our static method so that the first parameter—that is, the data type we are extending—is prefixed with the `this` keyword (see Listing 5.41).

LISTING 5.41: Static Copy Method for `DirectoryInfo`

```
public static class DirectoryInfoExtension
{
  public static void CopyTo(
      this DirectoryInfo sourceDirectory, string target,
      SearchOption option, string searchPattern)
  {
      // ...
  }
}
```

```
  // ...
      DirectoryInfo directory = new DirectoryInfo(".\\Source");
      directory.CopyTo(".\\Target",
          SearchOption.AllDirectories, "*");
  // ...
```

With this simple addition to C# 3.0, it is now possible to add "instance methods" to any class, including classes that are not within the same assembly. The resultant CIL code, however, is identical to what the compiler creates when calling the extension method as a normal static method.

3.0

Extension method requirements are as follows.

- The first parameter corresponds to the type that the method extends or on which it operates.
- To designate the extension method, prefix the extended type with the `this` modifier.
- To access the method as an extension method, import the extending type's namespace via a `using` directive (or place the extending class in the same namespace as the calling code).

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

If the extension method signature matches a signature already found on the extended type (that is, if `CopyTo()` already existed on `DirectoryInfo`), the extension method will never be called except as a normal static method.

Note that specializing a type via inheritance (covered in detail in Chapter 6) is preferable to using an extension method. Extension methods do not provide a clean versioning mechanism, because the addition of a matching signature to the extended type will take precedence over the extension method without warning of the change. The subtlety of this behavior is more pronounced for extended classes whose source code you don't control. Another minor point is that, although development IDEs support IntelliSense for extension methods, simply reading through the calling code does not make it obvious that a method is an extension method.

In general, you should use extension methods sparingly. Do not, for example, define them on type `object`. Chapter 6 discusses how to use extension methods in association with an interface. Without such an association, defining extension methods is rare.

Guidelines

AVOID frivolously defining extension methods, especially on types you don't own.

End 3.0

Encapsulating the Data

In addition to properties and the access modifiers we looked at earlier in the chapter, there are several other specialized ways of encapsulating the data within a class. For instance, there are two more field modifiers. The first is the `const` modifier, which you already encountered when declaring local variables. The second is the capability of fields to be defined as read-only.

const

Just as with `const` values, a `const` field contains a compile-time–determined value that cannot be changed at runtime. Values such as pi make good candidates for constant field declarations. Listing 5.42 shows an example of declaring a `const` field.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

LISTING 5.42: Declaring a Constant Field

```
class ConvertUnits
{
   public const float CentimetersPerInch = 2.54F;
   public const int CupsPerGallon = 16;
   // ...
}
```

Constant fields are static automatically, since no new field instance is required for each object instance. Declaring a constant field as `static` explicitly will cause a compile error. Also, constant fields are usually declared only for types that have literal values (`string`, `int`, and `double`, for example). Types such as `Program` or `System.Guid` cannot be used for constant fields.

It is important that the types of values used in `public` constant expressions are permanent in time. Values such as pi, Avogadro's number, and the circumference of the Earth are good examples. However, values that could potentially change over time are not. Build numbers, population counts, and exchange rates would be poor choices for constants.

Guidelines

DO use constant fields for values that will never change.

DO NOT use constant fields for values that will change over time.

ADVANCED TOPIC

Public Constants Should Be Permanent Values

Publicly accessible constants should be permanent, because changing the value of a constant will not necessarily take effect in the assemblies that use it. If an assembly references a constant from a different assembly, the value of the constant is compiled directly into the referencing assembly. Therefore, if the value in the referenced assembly is changed but the referencing assembly is not recompiled, the referencing assembly will still use the original value, not the new value. Values that could potentially change in the future should be specified as `readonly` instead.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

readonly

Unlike `const`, the `readonly` modifier is available only for fields (not for local variables). It declares that the field value is modifiable only from inside the constructor or via an initializer during declaration. Listing 5.43 demonstrates how to declare a read-only field.

LISTING 5.43: Declaring a Field As `readonly`

```
class Employee
{
  public Employee(int id)
  {
      _Id = id;
  }

  // ...

  public readonly int _Id;
  public int Id
  {
      get { return _Id; }
  }

  // Error: A readonly field cannot be assigned to (except
  // in a constructor or a variable initializer)
  // public void SetId(int id) =>
  //          _Id = id;

  // ...
}
```

Unlike constant fields, `readonly`-decorated fields can vary from one instance to the next. In fact, a read-only field's value can change within the constructor. Furthermore, read-only fields occur as either instance or static fields. Another key distinction is that you can assign the value of a read-only field at execution time rather than just at compile time. Given that read-only fields must be set in the constructor or initializer, such fields are the one case where the compiler requires the fields be accessed from code outside their corresponding property. Besides this one exception, you should avoid accessing a backing field from anywhere other than its wrapping property.

Another important feature of `readonly`-decorated fields over `const` fields is that read-only fields are not limited to types with literal values. It is possible, for example, to declare a `readonly System.Guid` instance field:

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
public static readonly Guid ComIUnknownGuid =
      new Guid("00000000-0000-0000-C000-000000000046");
```

The same, however, is not possible using a constant because of the fact that there is no C# literal representation of a Guid.

Begin 6.0

Given the guideline that fields should not be accessed from outside their wrapping property, those programming in a C# 6.0 world will discover that that there is almost never a need to use the `readonly` modifier. Instead, it is preferable to use a read-only automatically implemented property, as discussed earlier in the chapter.

Consider Listing 5.44 for one more read-only example.

LISTING 5.44: Declaring a Read-Only Automatically Implemented Property

```
class TicTacToeBoard
{
   // Set both player's move to all false (blank).
   //    |   |
   // ---+---+---
   //    |   |
   // ---+---+---
   //    |   |
      public bool[,,] Cells { get; } = new bool[2, 3, 3];
   // Error: The property Cells cannot
   // be assigned to because it is read-only
   public void SetCells(bool[,,] value) =>
            Cells = new bool[2, 3, 3];

  // ...
}
```

Whether implemented using C# 6.0 read-only automatically implemented properties or the `readonly` modifier on a field, providing for immutability of the array reference is a useful defensive coding technique. It ensures that the array instance remains the same, while allowing the elements within the array to change. Without the read-only constraint, it would be all too easy to mistakenly assign a new array to the member, thereby discarding the existing array rather than updating individual array elements. In other words, using a read-only approach with an array does not freeze the contents of the array. Rather, it freezes the array instance (and therefore the number of elements in the array) because it is not possible to reassign the value to a new instance. The elements of the array are still writeable.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Guidelines

DO favor use of read-only automatically implemented properties in C# 6.0 (or later) over defining read-only fields.

DO use `public static readonly` modified fields for predefined object instances prior to C# 6.0.

AVOID changing a public `readonly` modified field in pre-C# 6.0 to a read-only automatically implemented property in C# 6.0 (or later) if version API compatibility is required.

End 6.0

Nested Classes

In addition to defining methods and fields within a class, it is possible to define a class within a class. Such classes are called **nested classes.** You use a nested class when the class makes little sense outside the context of its containing class.

Consider a class that handles the command-line options of a program. Such a class is generally unique to each program, so there is no reason to make a `CommandLine` class accessible from outside the class that contains `Main()`. Listing 5.45 demonstrates such a nested class.

LISTING 5.45: Defining a Nested Class

```
// CommandLine is nested within Program
class Program
{
  // Define a nested class for processing the command line.
  private class CommandLine
  {
      public CommandLine(string[] arguments)
      {
          for(int argumentCounter=0;
              argumentCounter<arguments.Length;
              argumentCounter++)
          {
              switch (argumentCounter)
              {
                  case 0:
                      Action = arguments[0].ToLower();
                      break;
                  case 1:
                      Id = arguments[1];
                      break;
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
                case 2:
                    FirstName = arguments[2];
                    break;
                case 3:
                    LastName = arguments[3];
                    break;
            }
        }
    }
    public string Action;
    public string Id;
    public string FirstName;
    public string LastName;
}

static void Main(string[] args)
{
  CommandLine commandLine = new CommandLine(args);

  switch (commandLine.Action)
  {
      case "new":
          // Create a new employee
          // ...
          break;
      case "update":
          // Update an existing employee's data
          // ...
          break;
      case "delete":
          // Remove an existing employee's file.
          // ...
          break;
      default:
          Console.WriteLine(
              "Employee.exe " +
       "new|update|delete <id> [firstname] [lastname]");
          break;
  }
 }
}
```

The nested class in this example is `Program.CommandLine`. As with all class members, no containing class identifier is needed from inside the containing class, so you can simply refer to it as `CommandLine`.

One unique characteristic of nested classes is the ability to specify `private` as an access modifier for the class itself. Because the purpose of this class is to parse the command line and place each argument into a

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

separate field, `Program.CommandLine` is relevant only to the `Program` class in this application. The use of the `private` access modifier defines the intended accessibility of the class and prevents access from outside the class. You can do this only if the class is nested.

The `this` member within a nested class refers to an instance of the nested class, not the containing class. One way for a nested class to access an instance of the containing class is if the containing class instance is explicitly passed, such as via a constructor or method parameter.

Another interesting characteristic of nested classes is that they can access any member on the containing class, including private members. The converse is not true, however: It is not possible for the containing class to access a private member of the nested class.

Nested classes are rare. They should not be defined if they are likely to be referenced outside the containing type. Furthermore, treat `public` nested classes with suspicion; they indicate potentially poor code that is likely to be confusing and hard to discover.

Guidelines

AVOID publicly exposed nested types. The only exception is if the declaration of such a type is unlikely or pertains to an advanced customization scenario.

Language Contrast: Java—Inner Classes

Java includes not only the concept of a nested class, but also the concept of an inner class. Inner classes correspond to objects that are associated with the containing class instance rather than just a syntactic relationship. In C#, you can achieve the same structure by including an instance field of a nested type within the outer class. A factory method or constructor can ensure a reference to the corresponding instance of the outer class is set within the inner class instance as well.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Begin 2.0

Partial Classes

Another language feature added in C# 2.0 is **partial classes.** Partial classes are portions of a class that the compiler can combine to form a complete class. Although you could define two or more partial classes within the same file, the general purpose of a partial class is to allow the splitting of a class definition across multiple files. Primarily this is useful for tools that are generating or modifying code. With partial classes, the tools can work on a file separate from the one the developer is manually coding.

Defining a Partial Class

C# 2.0 (and later) allows declaration of a partial class by prepending a contextual keyword, `partial`, immediately before `class`, as Listing 5.46 shows.

LISTING 5.46: Defining a Partial Class

```
// File: Program1.cs
partial class Program
{
}
```

```
// File: Program2.cs
partial class Program
{
}
```

In this case, each portion of `Program` is placed into a separate file, as identified by the comment.

Besides their use with code generators, another common use of partial classes is to place any nested classes into their own files. This is in accordance with the coding convention that places each class definition within its own file. For example, Listing 5.47 places the `Program.CommandLine` class into a file separate from the core `Program` members.

LISTING 5.47: Defining a Nested Class in a Separate Partial Class

```
// File: Program.cs
partial class Program
{
  static void Main(string[] args)
  {
    CommandLine commandLine = new CommandLine(args);

    switch (commandLine.Action)
```

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
    {
       // ...
    }
  }
}
```

```
// File: Program+CommandLine.cs
partial class Program
{
  // Define a nested class for processing the command line.
  private class CommandLine
  {
     // ...
  }
}
```

Partial classes do not allow for extending compiled classes, or classes in other assemblies. They are simply a means of splitting a class implementation across multiple files within the same assembly.

End 2.0

Partial Methods

Begin 3.0

Beginning with C# 3.0, the language designers added the concept of partial methods, extending the partial class concept of C# 2.0. Partial methods are allowed only within partial classes, and like partial classes, their primary purpose is to accommodate code generation.

Consider a code generation tool that generates the `Person.Designer.cs` file for the `Person` class based on a `Person` table within a database. This tool examines the table and creates properties for each column in the table. The problem, however, is that frequently the tool cannot generate any validation logic that may be required because this logic is based on business rules that are not embedded into the database table definition. To overcome this difficulty, the developer of the `Person` class needs to add the validation logic. It is undesirable to modify `Person.Designer.cs` directly, because if the file is regenerated (to accommodate an additional column in the database, for example), the changes would be lost. Instead, the structure of the code for `Person` needs to be separated out so that the generated code appears in one file and the custom code (with business rules) is placed into a separate file unaffected by any regeneration. As we saw in the preceding section, partial classes are well suited for the task of splitting a class across multiple files, but they are not always sufficient. In many cases, we also need **partial methods**

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

Partial methods allow for a declaration of a method without requiring an implementation. However, when the optional implementation is included, it can be located in one of the sister partial class definitions, likely in a separate file. Listing 5.48 shows the partial method declaration and the implementation for the `Person` class.

LISTING 5.48: Defining a Nested Class in a Separate Partial Class

```
// File: Person.Designer.cs
public partial class Person
{
    #region Extensibility Method Definitions
    partial void OnLastNameChanging(string value);
    partial void OnFirstNameChanging(string value);
    #endregion

    // ...
    public System.Guid PersonId
    {
        // ...
    }
    private System.Guid _PersonId;

    // ...
    public string LastName
    {
        get
        {
            return _LastName;
        }
        set
        {
            if ((_LastName != value))
            {
                OnLastNameChanging(value);
                _LastName = value;
            }
        }
    }
    private string _LastName;

    // ...
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            if ((_FirstName != value))
```

3.0

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

```
            {
                OnFirstNameChanging(value);
                _FirstName = value;
            }
        }
    }
    private string _FirstName;

}
```

```
// File: Person.cs
partial class Person
{
  partial void OnLastNameChanging(string value)
  {
      if (value == null)
      {
          throw new ArgumentNullException("value");
      }
      if(value.Trim().Length == 0)
      {
          throw new ArgumentException(
              "LastName cannot be empty.",
              "value");
      }
  }
}
```

In the listing of `Person.Designer.cs` are declarations for the `OnLastNameChanging()` and `OnFirstNameChanging()` methods. Furthermore, the properties for the last and first names make calls to their corresponding changing methods. Even though the declarations of the changing methods contain no implementation, this code will successfully compile. The key is that the method declarations are prefixed with the contextual keyword `partial` in addition to the class that contains such methods.

3.0

In Listing 5.48, only the `OnLastNameChanging()` method is implemented. In this case, the implementation checks the suggested new `LastName` value and throws an exception if it is not valid. Notice that the signatures for `OnLastNameChanging()` between the two locations match.

Any partial method must return `void`. If the method didn't return `void` and the implementation was not provided, what would the expected return be from a call to a nonimplemented method? To avoid any invalid assumptions about the return, the C# designers decided to prohibit methods with returns other than `void`. Similarly, `out` parameters are not allowed on partial methods. If a return value is required, `ref` parameters may be used.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)

In summary, partial methods allow generated code to call methods that have not necessarily been implemented. Furthermore, if there is no implementation provided for a partial method, no trace of the partial method appears in the CIL. This helps keep code size small while keeping flexibility high.

End 3.0

SUMMARY

This chapter explained C# constructs for classes and object orientation in C#. Its coverage included a discussion of fields, and a discussion of how to access them on a class instance.

This chapter also discussed the key decision of whether to store data on a per-instance basis or across all instances of a type. Static data is associated with the class, and instance data is stored on each object.

In addition, the chapter explored encapsulation in the context of access modifiers for methods and data. The C# construct of properties was introduced, and you saw how to use it to encapsulate private fields.

The next chapter focuses on how to associate classes with each other via inheritance, and explores the benefits derived from this object-oriented construct.

This chapter is published with permission from Pearson publishing as Addison-Wesley from the book Essential C# 6 (ISBN 978-0134141046)