

05

Building Your Own Types with Object-Oriented Programming

This chapter is about making your own types using **object-oriented programming** (OOP). You will learn about all the different categories of members that a type can have, including fields to store data and methods to perform actions. You will use OOP concepts such as aggregation and encapsulation. You will also learn about language features such as tuple syntax support, out variables, inferred tuple names, and default literals.

This chapter will cover the following topics:

- Talking about OOP
- Building class libraries
- Storing data with fields
- Writing and calling methods
- Controlling access with properties and indexers
- Pattern matching with objects
- Working with records

Talking about object-oriented programming

An object in the real world is a thing, such as a car or a person, whereas an object in programming often represents something in the real world, such as a product or bank account, but this can also be something more abstract.

In C#, we use the `class` (mostly) or `struct` (sometimes) C# keywords to define a type of object. You will learn about the difference between classes and structs in *Chapter 6, Implementing Interfaces and Inheriting Classes*. You can think of a type as being a blueprint or template for an object.

The concepts of object-oriented programming are briefly described here:

- **Encapsulation** is the combination of the data and actions that are related to an object. For example, a `BankAccount` type might have data, such as `Balance` and `AccountName`, as well as actions, such as `Deposit` and `Withdraw`. When encapsulating, you often want to control what can access those actions and the data, for example, restricting how the internal state of an object can be accessed or modified from the outside.
- **Composition** is about what an object is made of. For example, a car is composed of different parts, such as four wheels, several seats, and an engine.
- **Aggregation** is about what can be combined with an object. For example, a person is not part of a car object, but they could sit in the driver's seat and then becomes the car's driver — two separate objects that are aggregated together to form a new component.
- **Inheritance** is about reusing code by having a subclass derive from a base or superclass. All functionality in the base class is inherited by and becomes available in the derived class. For example, the **base** or **super** `Exception` class has some members that have the same implementation across all exceptions, and the sub or derived `SqlException` class inherits those members and has extra members only relevant to when a SQL database exception occurs, like a property for the database connection.
- **Abstraction** is about capturing the core idea of an object and ignoring the details or specifics. C# has an `abstract` keyword which formalizes the concept. If a class is not explicitly `abstract`, then it can be described as being concrete. Base or superclasses are often `abstract`, for example, the superclass `Stream` is `abstract` and its subclasses, like `FileStream` and `MemoryStream`, are concrete. Only concrete classes can be used to create objects; `abstract` classes can only be used as the base for other classes because they are missing some implementation. Abstraction is a tricky balance. If you make a class more `abstract`, more classes will be able to inherit from it, but at the same time, there will be less functionality to share.
- **Polymorphism** is about allowing a derived class to override an inherited action to provide custom behavior.

Building class libraries

Class library assemblies group types together into easily deployable units (DLL files). Apart from when you learned about unit testing, you have only created console applications to contain your code. To make the code that you write reusable across multiple projects, you should put it in class library assemblies, just like Microsoft does.

Creating a class library

The first task is to create a reusable .NET class library:

1. In your existing `Code` folder, create a folder named `Chapter05`, with a subfolder named `PacktLibrary`.

2. In Visual Studio Code, navigate to **File | Save Workspace As...**, enter the name **Chapter05**, select the **Chapter05** folder, and click **Save**.
3. Navigate to **File | Add Folder to Workspace...**, select the **PacktLibrary** folder, and click **Add**.
4. In **TERMINAL**, enter the following command: `dotnet new classlib`.
5. Open the **PacktLibrary.csproj** file, and note that by default class libraries target .NET 5 and therefore can only work with other .NET 5-compatible assemblies, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

6. Modify the target framework to support .NET Standard 2.0, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

7. Save and close the file.
8. In **TERMINAL**, compile the project, as shown in the following command: `dotnet build`.



Good Practice: To use the latest C# language and .NET platform features, put types in a .NET 5 class library. To support legacy .NET platforms like .NET Core, .NET Framework, and Xamarin, put types that you might reuse in a .NET Standard 2.0 class library.

Defining a class

The next task is to define a class that will represent a person:

1. In **EXPLORER**, rename the file named **Class1.cs** to **Person.cs**.
2. Click **Person.cs** to open it and change the class name to **Person**.
3. Change the namespace to **Packt.Shared**.



Good Practice: We're doing this because it is important to put your classes in a logically named namespace. A better namespace name would be domain-specific, for example, `System.Numerics` for types related to advanced numbers, but in this case, the types we will create are `Person`, `BankAccount`, and `WondersOfTheWorld` and they do not have a normal domain.

Your class file should now look like the following code:

```
using System;

namespace Packt.Shared
{
    public class Person
    {
    }
}
```

Note that the C# keyword `public` is applied before `class`. This keyword is called an **access modifier**, and it allows for all the other code to access this class.

If you do not explicitly apply the `public` keyword, then it will only be accessible within the assembly that defined it. This is because the implicit access modifier for a class is `internal`. We need this class to be accessible outside the assembly, so we must make sure it is `public`.

Understanding members

This type does not yet have any members encapsulated within it. We will create some over the following pages. Members can be fields, methods, or specialized versions of both. You'll find a description of them here:

- **Fields** are used to store data. There are also three specialized categories of field, as shown in the following bullets:
 - **Constant:** The data never changes. The compiler literally copies the data into any code that reads it.
 - **Read-only:** The data cannot change after the class is instantiated, but the data can be calculated or loaded from an external source at the time of instantiation.
 - **Event:** The data references one or more methods that you want to execute when something happens, such as clicking on a button, or responding to a request from other code. Events will be covered in *Chapter 6, Implementing Interfaces and Inheriting Classes*.

- **Methods** are used to execute statements. You saw some examples when you learned about functions in *Chapter 4, Writing, Debugging, and Testing Functions*. There are also four specialized categories of method:
 - **Constructor:** The statements execute when you use the `new` keyword to allocate memory and instantiate a class.
 - **Property:** The statements execute when you get or set data. The data is commonly stored in a field but could be stored externally, or calculated at runtime. Properties are the preferred way to encapsulate fields unless the memory address of the field needs to be exposed.
 - **Indexer:** The statements execute when you get or set data using array syntax `[]`.
 - **Operator:** The statements execute when you use an operator like `+` and `/` on operands of your type.

Instantiating a class

In this section, we will make an **instance** of the `Person` class, which is described as instantiating a class.

Referencing an assembly

Before we can instantiate a class, we need to reference the assembly that contains it:

1. Create a subfolder under `Chapter05` named `PeopleApp`.
2. In **Visual Studio Code**, navigate to **File | Add Folder to Workspace...**, select the `PeopleApp` folder, and click **Add**.
3. Navigate to **Terminal | New Terminal** and select **PeopleApp**.
4. In **TERMINAL**, enter the following command: `dotnet new console`.
5. In **EXPLORER**, click on the file named `PeopleApp.csproj`.
6. Add a project reference to `PacktLibrary`, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\PacktLibrary\PacktLibrary.csproj" />
  </ItemGroup>
</Project>
```

7. In **TERMINAL**, enter a command to compile the PeopleApp project and its dependency PacktLibrary project, as shown in the following command: `dotnet build`.
8. Select PeopleApp as the active project for OmniSharp.

Importing a namespace to use a type

Now, we are ready to write statements to work with the Person class:

1. In **Visual Studio Code**, in the PeopleApp folder, open Program.cs.
2. At the top of the Program.cs file, enter statements to import the namespace for our People class and statically import the Console class, as shown in the following code:

```
using Packt.Shared;
using static System.Console;
```

3. In the Main method, enter statements to:
 - Create an instance of the Person type.
 - Output the instance using a textual description of itself.

The new keyword allocates memory for the object and initializes any internal data. We could use Person in place of the var keyword, but the use of var involves less typing and is still just as clear, as shown in the following code:

```
var bob = new Person();
WriteLine(bob.ToString());
```

You might be wondering, "Why does the bob variable have a method named ToString? The Person class is empty!" Don't worry, we're about to find out!

4. Run the application, by entering `dotnet run` in **TERMINAL**, and then view the result, as shown in the following output:

```
Packt.Shared.Person
```

Managing multiple files

If you have multiple files that you want to work with at the same time, then you can put them side by side as you edit them:

1. In **EXPLORER**, expand the two projects.
2. Open both Person.cs and Program.cs.
3. Click, hold, and drag the edit window tab for one of your open files to arrange them so that you can see both Person.cs and Program.cs at the same time.

You can click on the **Split Editor Right** button or press *Cmd + * so that you have two files open side by side vertically.



More Information: You can read more about working with the Visual Studio Code user interface at the following link: <https://code.visualstudio.com/docs/getstarted/userinterface>.

Understanding objects

Although our `Person` class did not explicitly choose to inherit from a type, all types ultimately inherit directly or indirectly from a special type named `System.Object`.

The implementation of the `ToString` method in the `System.Object` type simply outputs the full namespace and type name.

Back in the original `Person` class, we could have explicitly told the compiler that `Person` inherits from the `System.Object` type, as shown in the following code:

```
public class Person : System.Object
```

When class B **inherits** from class A, we say that A is the **base** or **superclass** and B is the **derived** or **subclass**. In this case, `System.Object` is the base or superclass and `Person` is the derived or subclass.

You can also use the C# alias keyword `object`:

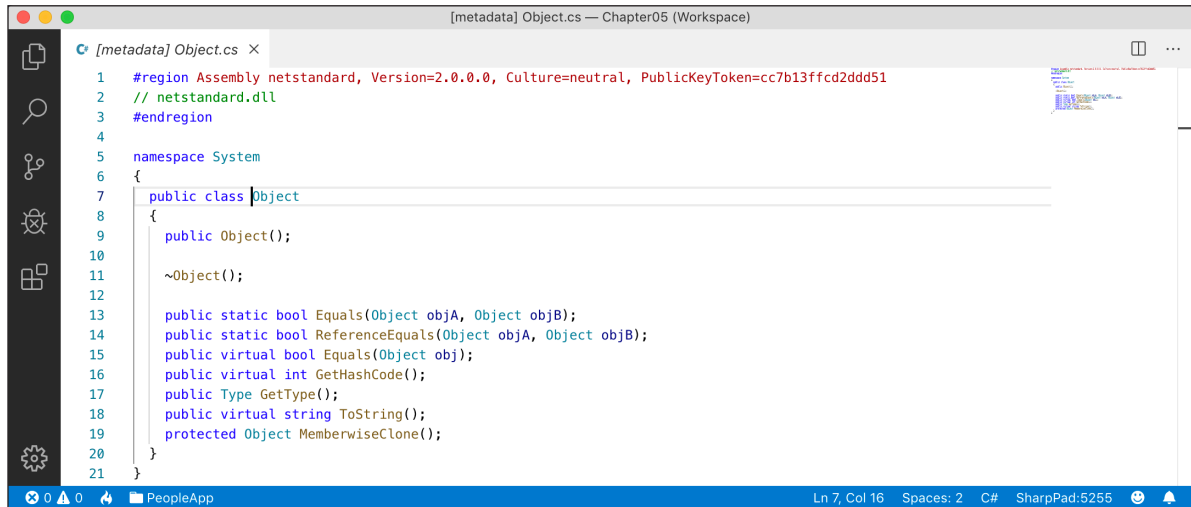
```
public class Person : object
```

Inheriting from System.Object

Let's make our class explicitly inherit from `object` and then review what members all objects have:

1. Modify your `Person` class to explicitly inherit from `object`.
2. Click inside the `object` keyword and press *F12*, or right-click on the `object` keyword and choose **Go to Definition**.

You will see the Microsoft-defined `System.Object` type and its members. This is something you don't need to understand the details of yet, but notice that it has a method named `ToString`, as shown in the following screenshot:



```
1 #region Assembly netstandard, Version=2.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51
2 // netstandard.dll
3 #endregion
4
5 namespace System
6 {
7     public class Object
8     {
9         public Object();
10
11         ~Object();
12
13         public static bool Equals(Object objA, Object objB);
14         public static bool ReferenceEquals(Object objA, Object objB);
15         public virtual bool Equals(Object obj);
16         public virtual int GetHashCode();
17         public Type GetType();
18         public virtual string ToString();
19         protected Object MemberwiseClone();
20     }
21 }
```



Good Practice: Assume other programmers know that if inheritance is not specified, the class will inherit from `System.Object`.

Storing data within fields

In this section, we will be defining a selection of fields in the class in order to store information about a person.

Defining fields

Let's say that we have decided that a person is composed of a name and a date of birth. We will encapsulate these two values inside a person, and the values will be visible outside it.

Inside the `Person` class, write statements to declare two public fields for storing a person's name and date of birth, as shown in the following code:

```
public class Person : object
{
    // fields
    public string Name;
    public DateTime DateOfBirth;
}
```


You can use any type for a field, including arrays and collections such as lists and dictionaries. These would be used if you needed to store multiple values in one named field. In this example, a person only has one name and one date of birth.

Understanding access modifiers

Part of encapsulation is choosing how visible the members are.

Note that, as we did with the class, we explicitly applied the `public` keyword to these fields. If we hadn't, then they would be implicitly `private` to the class, which means they are accessible only inside the class.

There are four access modifier keywords, and two combinations of access modifier keywords that you can apply to a class member, such as a field or method, as shown in the following table:

Access Modifier	Description
<code>private</code>	Member is accessible inside the type only. This is the default.
<code>internal</code>	Member is accessible inside the type and any type in the same assembly.
<code>protected</code>	Member is accessible inside the type and any type that inherits from the type.
<code>public</code>	Member is accessible everywhere.
<code>internal protected</code>	Member is accessible inside the type, any type in the same assembly, and any type that inherits from the type. Equivalent to a fictional access modifier named <code>internal_or_protected</code> .
<code>private protected</code>	Member is accessible inside the type or any type that inherits from the type and is in the same assembly. Equivalent to a fictional access modifier named <code>internal_and_protected</code> . This combination is only available with C# 7.2 or later.



Good Practice: Explicitly apply one of the access modifiers to all type members, even if you want to use the implicit access modifier for members, which is `private`. Additionally, fields should usually be `private` or `protected`, and you should then create `public` properties to get or set the field values. This is because it controls access. You will do this later in the chapter.

Setting and outputting field values

Now we will use those fields in the console app:

1. At the top of `Program.cs`, make sure the `System` namespace is imported.
2. Inside the `Main` method, change the statements to set the person's name and date of birth, and then output those fields nicely formatted, as shown in the following code:

```
var bob = new Person();
bob.Name = "Bob Smith";
bob.DateOfBirth = new DateTime(1965, 12, 22);
```

```
WriteLine(  
    format: "{0} was born on {1:dddd, d MMMM yyyy}",  
    arg0: bob.Name,  
    arg1: bob.DateOfBirth);
```

We could have used string interpolation too, but for long strings it will wrap over multiple lines, which can be harder to read in a printed book. In the code examples in this book, remember that `{0}` is a placeholder for `arg0`, and so on.

3. Run the application and view the result, as shown in the following output:

```
Bob Smith was born on Wednesday, 22 December 1965
```

The format code for `arg1` is made of several parts. `dddd` means the name of the day of the week. `d` means the number of the day of the month. `MMMM` means the name of the month. Lowercase `m` is used for minutes in time values. `yyyy` means the full number of the year. `yy` would mean the two-digit year.

You can also initialize fields using a shorthand object initializer syntax using curly braces. Let's see how.

4. Add the following code underneath the existing code to create another new person. Notice the different format code for the date of birth when writing to the console:

```
var alice = new Person  
{  
    Name = "Alice Jones",  
    DateOfBirth = new DateTime(1998, 3, 7)  
};  
  
WriteLine(  
    format: "{0} was born on {1:dd MMM yy}",  
    arg0: alice.Name,  
    arg1: alice.DateOfBirth);
```

5. Run the application and view the result, as shown in the following output:

```
Bob Smith was born on Wednesday, 22 December 1965  
Alice Jones was born on 07 Mar 98
```

Remember that your output may look different based on your locale, that is, language and culture.

Storing a value using an enum type

Sometimes, a value needs to be one of a limited set of options. For example, there are seven ancient wonders of the world, and a person may have one favorite. At other times, a value needs to be a combination of a limited set of options. For example, a person may have a bucket list of ancient world wonders they want to visit.

We are able to store this data by defining an enum type.

An enum type is a very efficient way of storing one or more choices because, internally, it uses integer values in combination with a lookup table of string descriptions:

1. Add a new file to the class library by selecting `PacktLibrary`, clicking on the **New File** button in the mini toolbar, and entering the name `WondersOfTheAncientWorld.cs`.
2. Modify the `WondersOfTheAncientWorld.cs` file, as shown in the following code:

```
namespace Packt.Shared
{
    public enum WondersOfTheAncientWorld
    {
        GreatPyramidOfGiza,
        HangingGardensOfBabylon,
        StatueOfZeusAtOlympia,
        TempleOfArtemisAtEphesus,
        MausoleumAtHalicarnassus,
        ColossusOfRhodes,
        LighthouseOfAlexandria
    }
}
```

3. In the `Person` class, add the following statement to your list of fields:

```
public WondersOfTheAncientWorld FavoriteAncientWonder;
```

4. In the `Main` method of `Program.cs`, add the following statements:

```
bob.FavoriteAncientWonder =
    WondersOfTheAncientWorld.StatueOfZeusAtOlympia;

WriteLine(format:
    "{0}'s favorite wonder is {1}. Its integer is {2}.",
    arg0: bob.Name,
    arg1: bob.FavoriteAncientWonder,
    arg2: (int)bob.FavoriteAncientWonder);
```

5. Run the application and view the result, as shown in the following output:

```
Bob Smith's favorite wonder is StatueOfZeusAtOlympia. Its integer is 2.
```

The enum value is internally stored as an `int` for efficiency. The `int` values are automatically assigned starting at 0, so the third world wonder in our enum has a value of 2. You can assign `int` values that are not listed in the enum. They will output as the `int` value instead of a name since a match will not be found.

Storing multiple values using an enum type

For the bucket list, we could create a collection of instances of the enum, and collections will be explained later in this chapter, but there is a better way. We can combine multiple choices into a single value using **flags**:

1. Modify the enum by decorating it with the `[System.Flags]` attribute.
2. Explicitly set a byte value for each wonder that represents different bit columns, as shown in the following code:

```
namespace Packt.Shared
{
    [System.Flags]
    public enum WondersOfTheAncientWorld : byte
    {
        None = 0b_0000_0000, // i.e. 0
        GreatPyramidOfGiza = 0b_0000_0001, // i.e. 1
        HangingGardensOfBabylon = 0b_0000_0010, // i.e. 2
        StatueOfZeusAtOlympia = 0b_0000_0100, // i.e. 4
        TempleOfArtemisAtEphesus = 0b_0000_1000, // i.e. 8
        MausoleumAtHalicarnassus = 0b_0001_0000, // i.e. 16
        ColossusOfRhodes = 0b_0010_0000, // i.e. 32
        LighthouseOfAlexandria = 0b_0100_0000 // i.e. 64
    }
}
```

We are assigning explicit values for each choice that would not overlap when looking at the bits stored in memory. We should also decorate the enum type with the `System.Flags` attribute so that when the value is returned it can automatically match with multiple values as a comma-separated string instead of returning an int value. Normally, an enum type uses an int variable internally, but since we don't need values that big, we can reduce memory requirements by 75%, that is, 1 byte per value instead of 4 bytes, by telling it to use a byte variable.

If we want to indicate that our bucket list includes the *Hanging Gardens* and *Mausoleum at Halicarnassus* ancient world wonders, then we would want the 16 and 2 bits set to 1. In other words, we would store the value 18:

64	32	16	8	4	2	1
0	0	1	0	0	1	0

In the `Person` class, add the following statement to your list of fields:

```
public WondersOfTheAncientWorld BucketList;
```

3. In the `Main` method of `PeopleApp`, add the following statements to set the bucket list using the `|` operator (bitwise logical OR) to combine the enum values. We could also set the value using the number 18 cast into the enum type, as shown in the comment, but we shouldn't because that would make the code harder to understand:

```

bob.BucketList =
    WondersOfTheAncientWorld.HangingGardensOfBabylon
    | WondersOfTheAncientWorld.MausoleumAtHalicarnassus;

// bob.BucketList = (WondersOfTheAncientWorld)18;

WriteLine($"{bob.Name}'s bucket list is {bob.BucketList}");

```

4. Run the application and view the result, as shown in the following output:

```

Bob Smith's bucket list is HangingGardensOfBabylon,
MausoleumAtHalicarnassus

```



Good Practice: Use the enum values to store combinations of discrete options. Derive an enum type from `byte` if there are up to eight options, from `ushort` if there are up to 16 options, from `uint` if there are up to 32 options, and from `ulong` if there are up to 64 options.

Storing multiple values using collections

Let's now add a field to store a person's children. This is an example of aggregation because children are instances of a class that is related to the current person but are not part of the person itself. We will use a generic `List<T>` collection type:

1. Import the `System.Collections.Generic` namespace at the top of the `Person.cs` class file, as shown in the following code:

```
using System.Collections.Generic;
```

You will learn more about collections in *Chapter 8, Working with Common .NET Types*. For now, just follow along.

2. Declare a new field in the `Person` class, as shown in the following code:

```
public List<Person> Children = new List<Person>();
```

`List<Person>` is read aloud as "list of Person," for example, "the type of the property named `Children` is a list of `Person` instances." We must ensure the collection is initialized to a new instance of a list of `Person` before we can add items to it, otherwise, the field will be `null` and it will throw runtime exceptions.

The angle brackets in the `List<T>` type is a feature of C# called **generics** that was introduced in 2005 with C# 2.0. It's just a fancy term for making a collection **strongly typed**, that is, the compiler knows more specifically what type of object can be stored in the collection. Generics improve the performance and correctness of your code.

Strongly typed is different from **statically typed**. The old `System.Collection` types are statically typed to contain weakly typed `System.Object` items. The newer `System.Collection.Generic` types are statically typed to contain strongly typed `<T>` instances. Ironically, the term **generics** means we can use a more specific static type!

1. In the `Main` method, add statements to add two children for Bob and then show how many children he has and what their names are, as shown in the following code:

```
bob.Children.Add(new Person { Name = "Alfred" });
bob.Children.Add(new Person { Name = "Zoe" });

WriteLine(
    $"{bob.Name} has {bob.Children.Count} children:");

for (int child = 0; child < bob.Children.Count; child++)
{
    WriteLine($"{bob.Children[child].Name}");
}
```

We could also use a `foreach` statement. As an extra challenge, change the `for` statement to output the same information using `foreach`.

2. Run the application and view the result, as shown in the following output:

```
Bob Smith has 2 children:
    Alfred
    Zoe
```

Making a field static

The fields that we have created so far have all been **instance** members, meaning that a different value of each field exists for each instance of the class that is created. The `bob` variable has a different `Name` value to `alice`.

Sometimes, you want to define a field that only has one value that is shared across all instances. These are called **static** members because fields are not the only members that can be static.

Let's see what can be achieved using static fields:

1. In the `PacktLibrary` project, add a new class file named `BankAccount.cs`.
2. Modify the class to give it three fields, two instance fields and one static field, as shown in the following code:

```
namespace Packt.Shared
{
    public class BankAccount
    {
        public string AccountName; // instance member
    }
}
```

```

    public decimal Balance; // instance member
    public static decimal InterestRate; // shared member
}
}

```

Each instance of `BankAccount` will have its own `AccountName` and `Balance` values, but all instances will share a single `InterestRate` value.

3. In `Program.cs` and its `Main` method, add statements to set the shared interest rate and then create two instances of the `BankAccount` type, as shown in the following code:

```

BankAccount.InterestRate = 0.012M; // store a shared value

var jonesAccount = new BankAccount();
jonesAccount.AccountName = "Mrs. Jones";
jonesAccount.Balance = 2400;

WriteLine(format: "{0} earned {1:C} interest.",
    arg0: jonesAccount.AccountName,
    arg1: jonesAccount.Balance * BankAccount.InterestRate);

var gerrierAccount = new BankAccount();
gerrierAccount.AccountName = "Ms. Gerrier";
gerrierAccount.Balance = 98;

WriteLine(format: "{0} earned {1:C} interest.",
    arg0: gerrierAccount.AccountName,
    arg1: gerrierAccount.Balance * BankAccount.InterestRate);

```

:C is a format code that tells .NET to use the currency format for the numbers. In *Chapter 8, Working with Common .NET Types*, you will learn how to control the culture that determines the currency symbol. For now, it will use the default for your operating system installation. I live in London, UK, hence my output shows British Pounds (£).

4. Run the application and view the additional output:

```

Mrs. Jones earned £28.80 interest.
Ms. Gerrier earned £1.18 interest.

```

Making a field constant

If the value of a field will never **ever** change, you can use the `const` keyword and assign a literal value at compile time:

1. In the `Person` class, add the following code:

```

// constants
public const string Species = "Homo Sapien";

```

- In the Main method, add a statement to write Bob's name and species to the console, as shown in the following code:

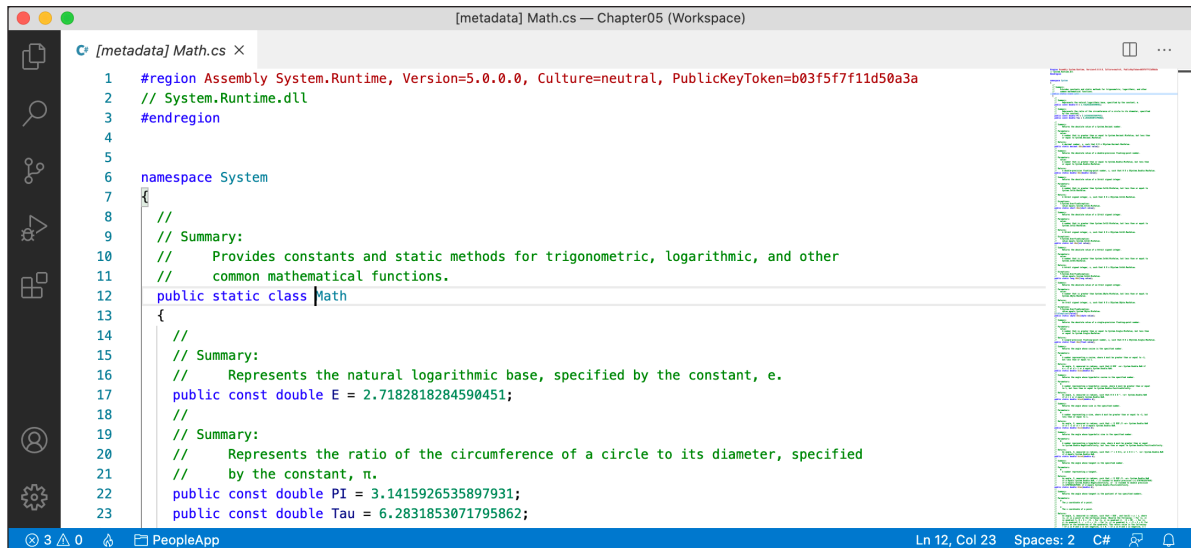
```
WriteLine($"{bob.Name} is a {Person.Species}");
```

To get the value of a constant field, you must write the name of the class, not the name of an instance of the class.

- Run the application and view the result, as shown in the following output:

```
Bob Smith is a Homo Sapien
```

Examples of the `const` fields in Microsoft types include `System.Int32.MaxValue` and `System.Math.PI` because neither value will ever change, as you can see in the following screenshot:



Good Practice: Constants should be avoided for two important reasons: the value must be known at compile time, and it must be expressible as a literal string, Boolean, or number value. Every reference to the `const` field is replaced with the literal value at compile time, which will, therefore, not be reflected if the value changes in a future version and you do not recompile any assemblies that reference it to get the new value.

Making a field read-only

A better choice for fields that should not change is to mark them as read-only:

- Inside the `Person` class, add a statement to declare an instance read-only field to store a person's home planet, as shown in the following code:

```
// read-only fields
public readonly string HomePlanet = "Earth";
```


You can also declare `static readonly` fields whose value will be shared across all instances of the type.

2. Inside the `Main` method, add a statement to write Bob's name and home planet to the console, as shown in the following code:

```
WriteLine($"{bob.Name} was born on {bob.HomePlanet}");
```

3. Run the application and view the result, as shown in the following output:

```
Bob Smith was born on Earth
```



Good Practice: Use read-only fields over constant fields for two important reasons: the value can be calculated or loaded at runtime and can be expressed using any executable statement. So, a read-only field can be set using a constructor or a field assignment. Every reference to the field is a live reference, so any future changes will be correctly reflected by calling code.

Initializing fields with constructors

Fields often need to be initialized at runtime. You do this in a constructor that will be called when you make an instance of the class using the `new` keyword. Constructors execute before any fields are set by the code that is using the type.

1. Inside the `Person` class, add the following highlighted code after the existing read-only `HomePlanet` field:

```
// read-only fields
public readonly string HomePlanet = "Earth";
public readonly DateTime Instantiated;

// constructors
public Person()
{
    // set default values for fields
    // including read-only fields
    Name = "Unknown";
    Instantiated = DateTime.Now;
}
```

2. Inside the `Main` method, add statements to instantiate a new person and then output its initial field values, as shown in the following code:

```
var blankPerson = new Person();

WriteLine(format:
    "{0} of {1} was created at {2:hh:mm:ss} on a {2:dddd}.",
    arg0: blankPerson.Name,
```

```
arg1: blankPerson.HomePlanet,  
arg2: blankPerson.Instantiated);
```

3. Run the application and view the result, as shown in the following output:

```
Unknown of Earth was created at 11:58:12 on a Sunday
```

You can have multiple constructors in a type. This is especially useful to encourage developers to set initial values for fields.

4. In the `Person` class, add statements to define a second constructor that allows a developer to set initial values for the person's name and home planet, as shown in the following code:

```
public Person(string initialName, string homePlanet)  
{  
    Name = initialName;  
    HomePlanet = homePlanet;  
    Instantiated = DateTime.Now;  
}
```

5. Inside the `Main` method, add the following code:

```
var gunny = new Person("Gunny", "Mars");  
  
WriteLine(format:  
    "{0} of {1} was created at {2:hh:mm:ss} on a {2:dddd}.",  
    arg0: gunny.Name,  
    arg1: gunny.HomePlanet,  
    arg2: gunny.Instantiated);
```

6. Run the application and view the result:

```
Gunny of Mars was created at 11:59:25 on a Sunday
```

Setting fields with default literals

A language feature introduced in C# 7.1 was **default literals**. Back in *Chapter 2, Speaking C#*, you learned about the `default(type)` keyword.

As a reminder, if you had some fields in a class that you wanted to initialize to their default type values in a constructor, you have been able to use `default(type)` since C# 2.0:

1. In the `PacktLibrary` folder, add a new file named `ThingOfDefaults.cs`.

2. In the `ThingOfDefaults.cs` file, add statements to declare a class with four fields of various types and set them to their default values in a constructor, as shown in the following code:

```
using System;
using System.Collections.Generic;

namespace Packt.Shared
{
    public class ThingOfDefaults
    {
        public int Population;
        public DateTime When;
        public string Name;
        public List<Person> People;

        public ThingOfDefaults()
        {
            Population = default(int); // C# 2.0 and later
            When = default(DateTime);
            Name = default(string);
            People = default(List<Person>);
        }
    }
}
```

You might think that the compiler ought to be able to work out what type we mean without being explicitly told, and you'd be right, but for the first 15 years of the C# compiler's life, it didn't. Finally, with the C# 7.1 and later compilers, it does.

3. Simplify the statements setting the defaults, as shown highlighted in the following code:

```
using System;
using System.Collections.Generic;

namespace Packt.Shared
{
    public class ThingOfDefaults
    {
        public int Population;
        public DateTime When;
        public string Name;
        public List<Person> People;

        public ThingOfDefaults()
        {
            Population = default; // C# 7.1 and later
        }
    }
}
```

```
        When = default;  
        Name = default;  
        People = default;  
    }  
}  
}
```

Constructors are a special category of method. Let's look at methods in more detail.

Writing and calling methods

Methods are members of a type that execute a block of statements.

Returning values from methods

Methods can return a single value or return nothing.

- A method that performs some actions but does not return a value indicates this with the `void` type before the name of the method.
- A method that performs some actions and returns a value indicates this with the type of the return value before the name of the method.

For example, you will create two methods:

- `WriteToConsole`: This will perform an action (writing some text to the console), but it will return nothing from the method, indicated by the `void` keyword.
- `GetOrigin`: This will return a text value, indicated by the `string` keyword.

Let's write the code:

1. Inside the `Person` class, statically import `System.Console`.
2. Add statements to define the two methods, as shown in the following code:

```
// methods  
public void WriteToConsole()  
{  
    WriteLine($"{Name} was born on a {DateOfBirth:dddd}.");  
}  
  
public string GetOrigin()  
{  
    return $"{Name} was born on {HomePlanet}.";  
}
```

3. Inside the `Main` method, add statements to call the two methods, as shown in the following code:

```
bob.WriteLine();  
WriteLine(bob.GetOrigin());
```

4. Run the application and view the result, as shown in the following output:

```
Bob Smith was born on a Wednesday.  
Bob Smith was born on Earth.
```

Combining multiple returned values using tuples

Each method can only return a single value that has a single type. That type could be a simple type, such as `string` in the previous example, a complex type, such as `Person`, or a collection type, such as `List<Person>`.

Imagine that we want to define a method named `GetTheData` that returns both a `string` value and an `int` value. We could define a new class named `TextAndNumber` with a `string` field and an `int` field, and return an instance of that complex type, as shown in the following code:

```
public class TextAndNumber  
{  
    public string Text;  
    public int Number;  
}  
  
public class Processor  
{  
    public TextAndNumber GetTheData()  
    {  
        return new TextAndNumber  
        {  
            Text = "What's the meaning of life?",  
            Number = 42  
        };  
    }  
}
```

But defining a class just to combine two values together is unnecessary, because in modern versions of C# we can use **tuples**. Tuples are an efficient way to combine two or more values into a single unit. I pronounce them as tuh-ples but I have heard other developers pronounce them as too-ples. To-may-toe, to-mah-toe, po-tay-toe, po-tah-toe, I guess.

Tuples have been a part of some languages such as F# since their first version, but .NET only added support for them in .NET 4.0 with the `System.Tuple` type.

It was only in C# 7.0 that C# added language syntax support for tuples using the parentheses characters `()` and at the same time, .NET added a new `System.ValueTuple` type that is more efficient in some common scenarios than the old .NET 4.0 `System.Tuple` type, and the C# tuple uses the more efficient one.

Let's explore tuples:

1. In the `Person` class, add statements to define a method that returns a string and int tuple, as shown in the following code:

```
public (string, int) GetFruit()
{
    return ("Apples", 5);
}
```

2. In the `Main` method, add statements to call the `GetFruit` method and then output the tuple's fields automatically named `Item1` and `Item2`, as shown in the following code:

```
(string, int) fruit = bob.GetFruit();

WriteLine($"{fruit.Item1}, {fruit.Item2} there are.");
```

3. Run the application and view the result, as shown in the following output:

```
Apples, 5 there are.
```

Naming the fields of a tuple

To access the fields of a tuple, the default names are `Item1`, `Item2`, and so on.

You can explicitly specify the field names:

1. In the `Person` class, add statements to define a method that returns a tuple with named fields, as shown in the following code:

```
public (string Name, int Number) GetNamedFruit()
{
    return (Name: "Apples", Number: 5);
}
```

2. In the `Main` method, add statements to call the method and output the tuple's named fields, as shown in the following code:

```
var fruitNamed = bob.GetNamedFruit();

WriteLine($"There are {fruitNamed.Number} {fruitNamed.Name}.");
```

3. Run the application and view the result, as shown in the following output:

```
There are 5 Apples.
```

Inferring tuple names

If you are constructing a tuple from another object, you can use a feature introduced in C# 7.1 called **tuple name inference**.

In the Main method, create two tuples, made of a string and int value each, as shown in the following code:

```
var thing1 = ("Neville", 4);
WriteLine($"{thing1.Item1} has {thing1.Item2} children.");

var thing2 = (bob.Name, bob.Children.Count);
WriteLine($"{thing2.Name} has {thing2.Count} children.");
```

In C# 7.0, both things would use the Item1 and Item2 naming schemes. In C# 7.1 and later, the second thing can infer the names Name and Count.

Deconstructing tuples

You can also deconstruct tuples into separate variables. The deconstructing declaration has the same syntax as named field tuples, but without a named variable for the tuple, as shown in the following code:

```
// store return value in a tuple variable with two fields
(string name, int age) tupleWithNamedFields = GetPerson();
// tupleWithNamedFields.name
// tupleWithNamedFields.age

// deconstruct return value into two separate variables
(string name, int age) = GetPerson();
// name
// age
```

This has the effect of splitting the tuple into its parts and assigning those parts to new variables.

1. In the Main method, add the following code:

```
(string fruitName, int fruitNumber) = bob.GetFruit();

WriteLine($"Deconstructed: {fruitName}, {fruitNumber}");
```

2. Run the application and view the result, as shown in the following output:

```
Deconstructed: Apples, 5
```



More Information: Deconstruction is not just for tuples. Any type can be deconstructed if it has a `Deconstruct` method. You can read about this at the following link: <https://docs.microsoft.com/en-us/dotnet/csharp/deconstruct>.

Defining and passing parameters to methods

Methods can have parameters passed to them to change their behavior. Parameters are defined a bit like variable declarations but inside the parentheses of the method, as the following shows:

1. In the `Person` class, add statements to define two methods, the first without parameters and the second with one parameter, as shown in the following code:

```
public string SayHello()
{
    return $"{Name} says 'Hello!'";
}

public string SayHelloTo(string name)
{
    return $"{Name} says 'Hello {name}!'";
}
```

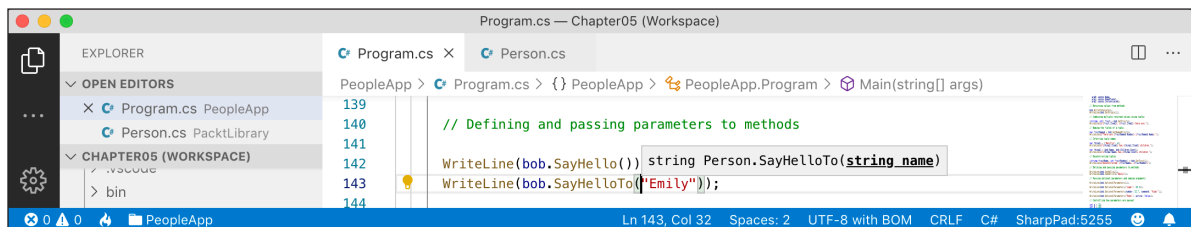
2. In the `Main` method, add statements to call the two methods and write the return value to the console, as shown in the following code:

```
WriteLine(bob.SayHello());
WriteLine(bob.SayHelloTo("Emily"));
```

3. Run the application and view the result:

```
Bob Smith says 'Hello!'
Bob Smith says 'Hello Emily!'
```

When typing a statement that calls a method, IntelliSense shows a tooltip with the name and type of any parameters, and the return type of the method, as shown in the following screenshot:

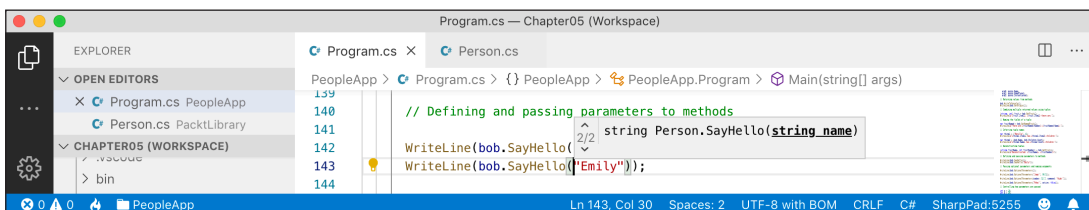


Overloading methods

Instead of having two different method names, we could give both methods the same name. This is allowed because the methods each have a different signature.

A **method signature** is a list of parameter types that can be passed when calling the method (as well as the type of the return value).

1. In the `Person` class, change the name of the `SayHelloTo` method to `SayHello`.
2. In `Main`, change the method call to use the `SayHello` method, and note that the quick info for the method tells you that it has one additional overload, `1/2`, as well as `2/2`, as shown in the following screenshot:



Good Practice: Use overloaded methods to simplify your class by making it appear to have fewer methods.

Passing optional parameters and naming arguments

Another way to simplify methods is to make parameters optional. You make a parameter optional by assigning a default value inside the method parameter list. Optional parameters must always come last in the list of parameters.



More Information: There is one exception to optional parameters always coming last. C# has a `params` keyword that allows you to pass a comma-separated list of parameters of any length as an array. You can read about `params` at the following link: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params>.

We will now create a method with three optional parameters.

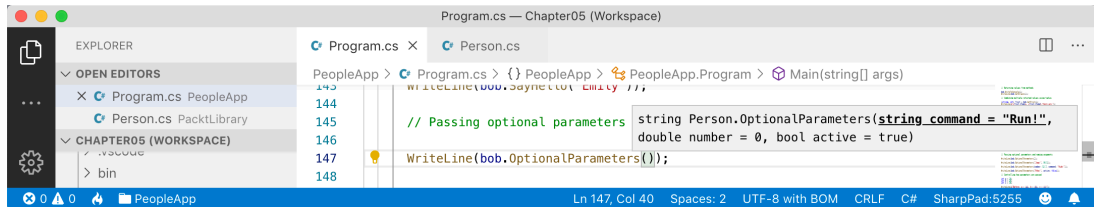
1. In the `Person` class, add statements to define the method, as shown in the following code:

```
public string OptionalParameters(
    string command = "Run!",
    double number = 0.0,
    bool active = true)
{
    return string.Format(
        format: "command is {0}, number is {1}, active is {2}",
        arg0: command, arg1: number, arg2: active);
}
```

2. In the Main method, add a statement to call the method and write its return value to the console, as shown in the following code:

```
WriteLine(bob.OptionalParameters());
```

3. Watch IntelliSense appear as you type the code. You will see a tooltip, showing the three optional parameters with their default values, as shown in the following screenshot:



4. Run the application and view the result, as shown in the following output:

```
command is Run!, number is 0, active is True
```

5. In the Main method, add a statement to pass a string value for the command parameter and a double value for the number parameter, as shown in the following code:

```
WriteLine(bob.OptionalParameters("Jump!", 98.5));
```

6. Run the application and see the result, as shown in the following output:

```
command is Jump!, number is 98.5, active is True
```

The default values for command and number have been replaced, but the default for active is still true.

Optional parameters are often combined with naming parameters when you call the method, because naming a parameter allows the values to be passed in a different order than how they were declared.

7. In the Main method, add a statement to pass a string value for the command parameter and a double value for the number parameter but using named parameters, so that the order they are passed through can be swapped around, as shown in the following code:

```
WriteLine(bob.OptionalParameters(  
    number: 52.7, command: "Hide!");
```

8. Run the application and view the result, as shown in the following output:

```
command is Hide!, number is 52.7, active is True
```

You can even use named parameters to skip over optional parameters.

9. In the Main method, add a statement to pass a string value for the command parameter using positional order, skip the number parameter, and use the named active parameter, as shown in the following code:

```
WriteLine(bob.OptionalParameters("Poke!", active: false));
```

10. Run the application and view the result, as shown in the following output:

```
command is Poke!, number is 0, active is False
```

Controlling how parameters are passed

When a parameter is passed into a method, it can be passed in one of three ways:

- By **value** (this is the default): Think of these as being *in-only*.
- By **reference** as a ref parameter: Think of these as being *in-and-out*.
- As an out parameter: Think of these as being *out-only*.

Let's see some examples of passing parameters in and out.

1. In the Person class, add statements to define a method with three parameters, one in parameter, one ref parameter, and one out parameter, as shown in the following method:

```
public void PassingParameters(int x, ref int y, out int z)
{
    // out parameters cannot have a default
    // AND must be initialized inside the method
    z = 99;

    // increment each parameter
    x++;
    y++;
    z++;
}
```

2. In the Main method, add statements to declare some int variables and pass them into the method, as shown in the following code:

```
int a = 10;
int b = 20;
int c = 30;

WriteLine($"Before: a = {a}, b = {b}, c = {c}");
bob.PassingParameters(a, ref b, out c);
WriteLine($"After: a = {a}, b = {b}, c = {c}");
```

3. Run the application and view the result, as shown in the following output:

```
Before: a = 10, b = 20, c = 30
After: a = 10, b = 21, c = 100
```

When passing a variable as a parameter by default, its current *value* gets passed, *not* the variable itself. Therefore, *x* is a copy of the *a* variable. The *a* variable retains its original value of 10. When passing a variable as a *ref* parameter, a *reference* to the variable gets passed into the method.

Therefore, *y* is a reference to *b*. The *b* variable gets incremented when the *y* parameter gets incremented. When passing a variable as an *out* parameter, a *reference* to the variable gets passed into the method.

Therefore, *z* is a reference to *c*. The *c* variable gets replaced by whatever code executes inside the method. We could simplify the code in the *Main* method by not assigning the value 30 to the *c* variable since it will always be replaced anyway.

In C# 7.0 and later, we can simplify code that uses the *out* variables.

4. In the *Main* method, add statements to declare some more variables including an *out* parameter named *f* declared inline, as shown in the following code:

```
int d = 10;
int e = 20;

WriteLine(
    $"Before: d = {d}, e = {e}, f doesn't exist yet!");

// simplified C# 7.0 syntax for the out parameter
bob.PassingParameters(d, ref e, out int f);
WriteLine($"After: d = {d}, e = {e}, f = {f}");
```

Understanding ref returns

In C# 7.0 and later, the *ref* keyword is not just for passing parameters into a method; it can also be applied to the return value. This allows an external variable to reference an internal variable and modify its value after the method call. This might be useful in advanced scenarios, for example, passing around placeholders into big data structures, but it's beyond the scope of this book.



More Information: You can read more about using the *ref* keyword for return values at the following link: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/ref-returns>.

Splitting classes using partial

When working on large projects with multiple team members, it is useful to be able to split the definition of a complex class across multiple files. You do this using the *partial* keyword.

Imagine we want to add statements to the `Person` class that are automatically generated by a tool like an object-relational mapper that reads schema information from a database. If the class is defined as `partial`, then we can split the class into an autogenerated code file and a manually edited code file.

1. In the `Person` class, add the `partial` keyword, as shown highlighted in the following code:

```
namespace Packt.Shared
{
    public partial class Person
    {
```

2. In **EXPLORER**, click on the **New File** button in the `PacktLibrary` folder, and enter a name of `PersonAutoGen.cs`.
3. Add statements to the new file, as shown in the following code:

```
namespace Packt.Shared
{
    public partial class Person
    {
    }
}
```

The rest of the code we write for this chapter will be written in the `PersonAutoGen.cs` file.

Controlling access with properties and indexers

Earlier, you created a method named `GetOrigin` that returned a string containing the name and origin of the person. Languages such as Java do this a lot. C# has a better way: **properties**.

A property is simply a method (or a pair of methods) that acts and looks like a field when you want to get or set a value, thereby simplifying the syntax.

Defining read-only properties

A readonly property only has a get implementation.

1. In the `PersonAutoGen.cs` file, in the `Person` class, add statements to define three properties:
 - The first property will perform the same role as the `GetOrigin` method using the property syntax that works with all versions of C# (although, it uses the C# 6 and later string interpolation syntax).
 - The second property will return a greeting message using the C# 6 and, later, the lambda expression (`=>`) syntax.
 - The third property will calculate the person's age.

Here's the code:

```
// a property defined using C# 1 - 5 syntax
public string Origin
{
    get
    {
        return $"{Name} was born on {HomePlanet}";
    }
}

// two properties defined using C# 6+ Lambda expression syntax
public string Greeting => $"{Name} says 'Hello!'";

public int Age => System.DateTime.Today.Year - DateOfBirth.Year;
```



More Information: Obviously, this isn't the best way to calculate someone's age, but we aren't learning how to calculate ages from dates of birth. If you need to do that properly, you can read a discussion at the following link: <https://stackoverflow.com/questions/9/how-do-i-calculate-someones-age-in-c>.

2. In the Main method, add statements to get the properties, as shown in the following code:

```
var sam = new Person
{
    Name = "Sam",
    DateOfBirth = new DateTime(1972, 1, 27)
};

WriteLine(sam.Origin);
WriteLine(sam.Greeting);
WriteLine(sam.Age);
```

3. Run the application and view the result, as shown in the following output:

```
Sam was born on Earth
Sam says 'Hello!'
48
```

The output shows 48 because I ran the console application on August 15, 2020 when Sam was 48 years old.

Defining settable properties

To create a settable property, you must use the older syntax and provide a pair of methods—not just a get part, but also a set part:

1. In the `PersonAutoGen.cs` file, add statements to define a string property that has both a get and set method (also known as a *getter* and *setter*), as shown in the following code:

```
public string FavoriteIceCream { get; set; } // auto-syntax
```

Although you have not manually created a field to store the person's favorite ice cream, it is there, automatically created by the compiler for you.

Sometimes, you need more control over what happens when a property is set. In this scenario, you must use a more detailed syntax and manually create a private field to store the value for the property.

2. In the `PersonAutoGen.cs` file, add statements to define a string field and string property that has both a get and set, as shown in the following code:

```
private string favoritePrimaryColor;

public string FavoritePrimaryColor
{
    get
    {
        return favoritePrimaryColor;
    }
    set
    {
        switch (value.ToLower())
        {
            case "red":
            case "green":
            case "blue":
                favoritePrimaryColor = value;
                break;
            default:
                throw new System.ArgumentException(
                    $"{value} is not a primary color. " +
                    "Choose from: red, green, blue.");
        }
    }
}
```

3. In the `Main` method, add statements to set Sam's favorite ice cream and color, and then write them to the console, as shown in the following code:

```

sam.FavoriteIceCream = "Chocolate Fudge";

WriteLine($"Sam's favorite ice-cream flavor is {sam.FavoriteIceCream}.");

sam.FavoritePrimaryColor = "Red";

WriteLine($"Sam's favorite primary color is {sam.FavoritePrimaryColor}.");

```

4. Run the application and view the result, as shown in the following output:

```

Sam's favorite ice-cream flavor is Chocolate Fudge.
Sam's favorite primary color is Red.

```

If you try to set the color to any value other than red, green, or blue, then the code will throw an exception. The calling code could then use a `try` statement to display the error message.



Good Practice: Use properties instead of fields when you want to validate what value can be stored when you want to data bind in XAML, which we will cover in *Chapter 21, Building Cross-Platform Mobile Apps*, and when you want to read and write to a field without using a method pair like `GetAge` and `SetAge`.



More Information: You can read more about the encapsulation of fields using properties at the following link: <https://stackoverflow.com/questions/1568091/why-use-getters-and-setters-accessors>.

Defining indexers

Indexers allow the calling code to use the array syntax to access a property. For example, the `string` type defines an **indexer** so that the calling code can access individual characters in the string individually.

We will define an indexer to simplify access to the children of a person:

1. In the `PersonAutoGen.cs` file, add statements to define an indexer to get and set a child using the index of the child, as shown in the following code:

```

// indexers
public Person this[int index]
{
    get
    {
        return Children[index];
    }
}

```



```

    set
    {
        Children[index] = value;
    }
}

```

You can overload indexers so that different types can be used for their parameters. For example, as well as passing an `int` value, you could also pass a `string` value.

2. In the `Main` method, add the following code. After adding to the children, we will access the first and second child using the longer `Children` field and the shorter indexer syntax:

```

sam.Children.Add(new Person { Name = "Charlie" });
sam.Children.Add(new Person { Name = "Ella" });

WriteLine($"Sam's first child is {sam.Children[0].Name}");
WriteLine($"Sam's second child is {sam.Children[1].Name}");

WriteLine($"Sam's first child is {sam[0].Name}");
WriteLine($"Sam's second child is {sam[1].Name}");

```

3. Run the application and view the result, as shown in the following output:

```

Sam's first child is Charlie
Sam's second child is Ella
Sam's first child is Charlie
Sam's second child is Ella

```

Pattern matching with objects

In *Chapter 3, Controlling Flow and Converting Types*, you were introduced to basic pattern matching. In this section, we will explore pattern matching in more detail.

Creating and referencing a .NET 5 class library

The enhanced pattern matching features are only available in .NET 5 class libraries that support C# 9 or later. First, we will see what pattern matching features were available before the enhancements in C# 9.

1. Create a subfolder under `Chapter05` named `PacktLibrary9`.
2. In Visual Studio Code, navigate to **File | Add Folder to Workspace...**, select the `PacktLibrary9` folder, and click **Add**.
3. Navigate to **Terminal | New Terminal** and select `PacktLibrary9`.
4. In **TERMINAL**, enter the following command: `dotnet new classlib`.
5. In **EXPLORER**, in the `PeopleApp` folder, click on the file named `PeopleApp.csproj`.

6. Add a language version element to force the use of the C# 8 compiler, and add a project reference to PacktLibrary9, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <LangVersion>8</LangVersion>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
    <ProjectReference Include="../PacktLibrary9/PacktLibrary9.csproj" />
  </ItemGroup>
</Project>
```

7. Navigate to **Terminal** | **New Terminal** and select **PeopleApp**.
8. In **TERMINAL**, enter a command to compile the PeopleApp project and its dependent projects, as shown in the following command: `dotnet build`.

Defining flight passengers

In this example, we will define some classes that represent various types of passengers on a flight and then we will use a switch expression with pattern matching to determine the cost of their flight.

1. In **EXPLORER**, in the PacktLibrary9 folder, delete the file named Class1.cs and add a new file named FlightPatterns.cs.
2. In the file FlightPatterns.cs, add statements to define three types of passengers with different properties, as shown in the following code:

```
namespace Packt.Shared
{
    public class BusinessClassPassenger
    {
        public override string ToString()
        {
            return $"Business Class";
        }
    }

    public class FirstClassPassenger
    {
        public int AirMiles { get; set; }

        public override string ToString()
        {

```

```

        return $"First Class with {AirMiles:N0} air miles";
    }
}

public class CoachClassPassenger
{
    public double CarryOnKG { get; set; }

    public override string ToString()
    {
        return $"Coach Class with {CarryOnKG:N2} KG carry on";
    }
}
}

```

3. In the PeopleApp folder, open Program.cs, and add statements to the end of the Main method to define an object array containing five passengers of various types and property values, and then enumerate them, outputting the cost of their flight, as shown in the following code:

```

object[] passengers = {
    new FirstClassPassenger { AirMiles = 1_419 },
    new FirstClassPassenger { AirMiles = 16_562 },
    new BusinessClassPassenger(),
    new CoachClassPassenger { CarryOnKG = 25.7 },
    new CoachClassPassenger { CarryOnKG = 0 },
};

foreach (object passenger in passengers)
{
    decimal flightCost = passenger switch
    {
        FirstClassPassenger p when p.AirMiles > 35000 => 1500M,
        FirstClassPassenger p when p.AirMiles > 15000 => 1750M,
        FirstClassPassenger _ => 2000M,
        BusinessClassPassenger _ => 1000M,
        CoachClassPassenger p when p.CarryOnKG < 10.0 => 500M,
        CoachClassPassenger _ => 650M,
        _ => 800M
    };

    WriteLine($"Flight costs {flightCost:C} for {passenger}");
}

```

While reviewing the preceding code, note the following:

- To pattern match on properties of an object, you must name a local variable that can then be used in an expression like `p`.
 - To pattern match on a type only, you can use the `_` to discard the local variable.
 - The switch expression also uses the `_` to represent its default branch.
4. Run the application and view the result, as shown in the following output:

```
Flight costs £2,000.00 for First Class with 1,419 air miles
Flight costs £1,750.00 for First Class with 16,562 air miles
Flight costs £1,000.00 for Business Class
Flight costs £650.00 for Coach Class with 25.70 KG carry on
Flight costs £500.00 for Coach Class with 0.00 KG carry on
```

Enhancements to pattern matching in C# 9

The previous examples worked with C# 8. Now we will look at some enhancements in C# 9 and later. First, you no longer need to use the underscore to discard when doing type matching:

1. In the `PeopleApp` folder, open `Program.cs` and remove the `_` from one of the branches.
2. In **TERMINAL**, enter the `dotnet build` command to compile the console app, and note the compile error that explains this feature is not supported by C# 8.0.
3. Open `PeopleApp.csproj` and remove the language version element that forced the use of C# 8.0.
4. In the `PeopleApp` folder, open `Program.cs` and modify the branches for first-class passengers to use a nested switch expression and the new support for conditionals like `>`, as shown in the following code:

```
decimal flightCost = passenger switch
{
    /* C# 8 syntax
    FirstClassPassenger p when p.AirMiles > 35000 => 1500M,
    FirstClassPassenger p when p.AirMiles > 15000 => 1750M,
    FirstClassPassenger                               => 2000M, */

    // C# 9 syntax
    FirstClassPassenger p => p.AirMiles switch
    {
        > 35000 => 1500M,
        > 15000 => 1750M,
        _      => 2000M
    },

    BusinessClassPassenger                => 1000M,
```

```
CoachClassPassenger p when p.CarryOnKG < 10.0 => 500M,
CoachClassPassenger                               => 650M,
-                                                    => 800M
};
```

5. Run the application, view the results, and note they are the same as before.



More Information: You can complete a detailed tutorial about pattern matching at the following link: <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/pattern-matching>.

Working with records

Before we dive into the new records language feature of C# 9, let us see some other related new features.

Init-only properties

You have used object initialization syntax to instantiate objects and set initial properties throughout this chapter. Those properties can also be changed after instantiation.

Sometimes you want to treat properties like readonly fields so they can be set during instantiation but not after. The new `init` keyword enables this. It can be used in place of the `set` keyword:

1. In the `PacktLibrary9` folder, add a new file named `Records.cs`.
2. In the `Records.cs` file, define an immutable person class, as shown in the following code:

```
namespace Packt.Shared
{
    public class ImmutablePerson
    {
        public string FirstName { get; init; }
        public string LastName { get; init; }
    }
}
```

3. In `Program.cs`, at the bottom of the `Main` method, add statements to instantiate a new immutable person and then try to change one of its properties, as shown in the following code:

```
var jeff = new ImmutablePerson
{
    FirstName = "Jeff",
```

```
        LastName = "Winger"  
    };  
  
    jeff.FirstName = "Geoff";
```

4. Compile the console app and note the compile error, as shown in the following output:

```
Program.cs(254,7): error CS8852: Init-only property or indexer  
'ImmutablePerson.FirstName' can only be assigned in an object initializer,  
or on 'this' or 'base' in an instance constructor or an 'init' accessor.  
[/Users/markjprice/Code/Chapter05/PeopleApp/PeopleApp.csproj]
```

5. Comment out the attempt to set the LastName property after instantiation.

Understanding records

Init-only properties provide some immutability to C#. You can take the concept further by using **Records**. These are defined by using the record keyword instead of the class keyword. That makes the whole object immutable, so it acts like a value.

Records should not have any state (properties and fields) that changes after instantiation. Instead, the idea is that you create new records from existing ones with any changed state. This is called **non-destructive mutation**. To do this, C# 9 introduces the with keyword:

1. Open Records.cs, and add a record named ImmutableVehicle, as shown highlighted in the following code:

```
public record ImmutableVehicle  
{  
    public int Wheels { get; init; }  
    public string Color { get; init; }  
    public string Brand { get; init; }  
}
```

2. Open Program.cs, and at the bottom of the Main method, add statements to create a car and then a mutated copy of it, as shown in the following code:

```
var car = new ImmutableVehicle  
{  
    Brand = "Mazda MX-5 RF",  
    Color = "Soul Red Crystal Metallic",  
    Wheels = 4  
};  
  
var repaintedCar = car with { Color = "Polymetal Grey Metallic" };  
  
WriteLine("Original color was {0}, new color is {1}.",  
    arg0: car.Color, arg1: repaintedCar.Color);
```

3. Run the application, view the results, and note the change to the car color in the mutated copy, as shown in the following output:

```
Original color was Soul Red Crystal Metallic, new color is Polymetal Grey Metallic.
```

Simplifying data members

In the following class, `Age` is a private field so it can only be accessed inside the class, as shown in the following code:

```
public class Person
{
    int Age; // private field by default
}
```

But with the `record` keyword, the field becomes an `init-only` public property, as shown in the following code:

```
public record Person
{
    int Age; // public property equivalent to:
    // public int Age { get; init; }
}
```

This is designed to make it clear and concise to define records.

Positional records

Instead of using object initialization syntax with curly braces, sometimes you might prefer to provide a constructor with positional parameters as you saw earlier in this chapter. You can also combine this with a deconstructor for splitting the object into individual parts, as shown in the following code:

```
public record ImmutableAnimal
{
    string Name; // i.e. public init-only properties
    string Species;

    public ImmutableAnimal(string name, string species)
    {
        Name = name;
        Species = species;
    }

    public void Deconstruct(out string name, out string species)
```

```
{
    name = Name;
    species = Species;
}
}
```

The properties, constructor, and deconstructor can be generated for you:

1. In the `Records.cs` file, add statements to define another record, as shown in the following code:

```
// simpler way to define a record that does the equivalent
public data class ImmutableAnimal(string Name, string Species);
```

2. In the `Program.cs` file, add statements to construct and deconstruct immutable animals, as shown in the following code:

```
var oscar = new ImmutableAnimal("Oscar", "Labrador");
var (who, what) = oscar; // calls Deconstruct method
WriteLine($"{who} is a {what}.");
```

3. Run the application and view the results, as shown in the following output:

```
Oscar is a Labrador.
```

Practicing and exploring

Test your knowledge and understanding by answering some questions, get some hands-on practice, and explore this chapter's topics with deeper research.

Exercise 5.1 – Test your knowledge

Answer the following questions:

1. What are the six combinations of access modifier keywords and what do they do?
2. What is the difference between the `static`, `const`, and `readonly` keywords when applied to a type member?
3. What does a constructor do?
4. Why should you apply the `[Flags]` attribute to an enum type when you want to store combined values?
5. Why is the `partial` keyword useful?
6. What is a tuple?
7. What does the C# record keyword do?
8. What does overloading mean?
9. What is the difference between a field and a property?
10. How do you make a method parameter optional?

Exercise 5.2 – Explore topics

Use the following links to read more about this chapter's topics:

- **Fields (C# programming guide):** <https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/classes-and-structs/fields>
- **Access modifiers (C# programming guide):** <https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/access-modifiers>
- **Enumeration types (C# reference):** <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum>
- **Constructors (C# programming guide):** <https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/classes-and-structs/constructors>
- **Methods (C# programming guide):** <https://docs.microsoft.com/en-us/dotnet/articles/csharp/methods>
- **Properties (C# programming guide):** <https://docs.microsoft.com/en-us/dotnet/articles/csharp/properties>

Summary

In this chapter, you learned about making your own types using OOP. You learned about some of the different categories of members that a type can have, including fields to store data and methods to perform actions, and you used OOP concepts, such as aggregation and encapsulation. You saw examples of how to use C# 9 features like object pattern matching enhancements, `init-only` properties, and records.

In the next chapter, you will take these concepts further by defining delegates and events, implementing interfaces, and inheriting from existing classes.

