CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 5

Talking about OOP

- **OOP**: 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 C# keywords **class**, **record**, and **struct** to define a type of object
- You can think of a type as being a blueprint or template for an object

Talking about OOP

- **Encapsulation**
  - The combination of the data and actions that are related to an object
  - E.g., 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
  - E.g., restricting how the internal state of an object can be accessed or modified from the outside

Talking about OOP

- **Composition**
  - What an object is made of
  - E.g., a Car is composed of different parts
    - four Wheel objects
    - several Seat objects
    - an Engine object

- **Aggregation**
  - What can be combined with an object
  - E.g., a Person is not part of a Car object
    - they could sit in the driver's Seat and then become the car's Driver
    - two separate objects that are aggregated together to form a new component

Talking about OOP

- **Inheritance**
  - Reusing code by having a **subclass** derive from a **base** or **superclass**
    - superclass, parent class, base class
    - subclass, child class, derived class
  - All functionality in the base class is inherited by and becomes available in the **derived** class
  - E.g., the base Exception class has some members that have the same implementation across all exceptions
    - the 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

Talking about OOP

- **Abstraction**
  - Capturing the core idea of an object and ignoring the details or specifics
  - C# has the abstract keyword that formalizes this concept
  - If a class is not explicitly **abstract**, then it can be described as being **concrete**
  - Base classes are often abstract
  - E.g., the base class **Stream** is abstract, and its derived classes, 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

Talking about OOP

- **Polymorphism**
  - Allowing a derived class to override an inherited action to provide custom behavior

Defining a class in a namespace

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

1. Add a new class file named Person.cs
2. Statically import System.Console
3. Set the namespace to Packt.Shared

- This is known as a file-scoped namespace declaration
- You can only have one file-scoped namespace per file

```
using System;
using static System.Console;

namespace Packt.Shared {    // namespace start

  public class Person {     // private by default
  }                         // internal by default

}                           // namespace end

// .NET 6.0 or later and C# 10 or later
using System;
// the class in this file is in this namespace
namespace Packt.Shared;

public class Person
{
}
```

Understanding members

- **Fields** are used to store data
  - **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
    - 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 some other code

Understanding members

- **Methods** are used to execute statements
  - **Constructor**:
    - Statements execute when you use the new keyword to allocate memory to instantiate a class
  - **Property**:
    - Statements execute when you get or set data
    - The data is commonly stored in a field
    - Properties are the preferred way to encapsulate fields
  - **Indexer**:
    - Statements execute when you get or set data using "array" syntax []
  - **Operator**:
    - Statements execute when you use an operator like + and / on operands

Defining fields

- Let's say that a person is composed of a name and a date and time of birth
- Encapsulate these two values inside a person – make visible outside class
- 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:

```
0 references
public class Person : object
{
    #region Fields: Data or state for this person
    0 references
    public string? Name; // ? means it can be null
    0 references
    public DateTimeOffset Born;
    #endregion
}
```

Types for fields

- 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

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

Setting field values using object initializer syntax

- You can also initialize fields using a shorthand **object initializer** syntax with curly braces

```
Person alice = new()
{
    Name = "Alice Jones",
    // This is an optional offset from UTC time zone
    Born = new(1998, 3, 7, 16, 28, 0, TimeSpan.Zero)
};

WriteLine(format: "{0} was born on {1:d}.", // Short date
    arg0: alice.Name, arg1: alice.Born);
```

```
Alice Jones was born on 3/7/1998.
```

Storing a value using an enum type

- Sometimes, a value needs to be one of a limited set of options
- E.g., 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
- E.g., 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
  - Internally, it uses integer values in combination with a lookup table of string descriptions

Storing a value using an enum type

```
namespace Packt.Shared;
2 references
public enum WondersOfTheAncientWorld {
    // constants - GREAT_PYRAMID_OF_GIZA,
    0 references
    GreatPyramidOfGiza,
    0 references
    HangingGardensOfBabylon,
    0 references
    StatueOfZeusAtOlympia,
    0 references
    TempleOfArtemisAtEphesus,
    0 references
    MausoleumAtHalicarnassus,
    0 references
    ColossusOfRhodes,
    0 references
    LighthouseOfAlexandria
}
0 references
public WondersOfTheAncientWorld FavoriteAncientWonder;

ken.FavoriteAncientWonder = WondersOfTheAncientWorld.GreatPyramidOfGiza;
```

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 that can store an ordered collection of any type

```
using System.Collections.Generic;                  // List<T>

public List<Person> children = new List<Person>();

public var children = new List<Person>();          // inferred typing

public List<Person> children = new();              // .NET 6.0
                                                   // target-typed new
```

Understanding generic collections

- The angle brackets in the List<T> type is a feature of C# called **generics**
- It's a fancy term for making a collection **strongly typed**, that is, the compiler knows specifically what type of object can be stored in the collection
- Generics improve the performance and correctness of your code
- **Strongly typed** has a different meaning to **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!

Understanding generic collections

```
// Works with all versions of C#.
0 references
Person alfred = new Person();
alfred.Name = "Alfred";
bob.Children.Add(alfred);

// Works with C# 3 and later
bob.Children.Add(new Person { Name = "Bella" });

// Works with C# 9 and later
bob.Children.Add(new() { Name = "Zoe" });
WriteLine($"{bob.Name} has {bob.Children.Count} children:");
```

Making a field static

- The fields 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 alice and bob variables have different Name values
- 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

Making a field static

```
namespace Packt.Shared;
1 reference
public class BankAccount {
    0 references
    public string? AccountName;          // Instance member. It could be null.
    0 references
    public decimal Balance;              // Instance member. Defaults to zero.
    0 references
    public static decimal InterestRate; // Shared member. Defaults to zero.
}
0 references
BankAccount jonesAccount = new();
jonesAccount.AccountName = "Mrs. Jones";
jonesAccount.Balance = 2400;
0 references
WriteLine(format: "{0} earned {1:C} interest.",
    arg0: jonesAccount.AccountName,
    arg1: jonesAccount.Balance * BankAccount.InterestRate);
```

Making a field constant or read-only

```
// constants
public const string Species = "Homo Sapiens";

// Often a better choice for fields that should not change
// is to mark them as read-only

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

- **Good Practice**:
- Use read-only fields over constant fields for two important reasons:
  - value can be calculated or loaded at runtime and can be expressed using any executable statement
    - 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 the 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

```
// 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;
}

Person blankPerson = new();

WriteLine(format:
    "{0} of {1} created {2:hh:mm:ss} on {2:dddd}.",
    arg0: blankPerson.Name,
    arg1: blankPerson.HomePlanet,
    arg2: blankPerson.Instantiated);
```

Defining multiple constructors

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

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

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

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

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

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

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

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
- Overloaded methods cannot differ only in the return type

Passing optional parameters

- 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

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

WriteLine(bob.OptionalParameters());
// command Run!, number 0, active True

WriteLine(bob.OptionalParameters("Jump!", 98.5));
//command Jump!, number 98.5, active True
```

Naming parameter values when calling methods

- 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

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

WriteLine(bob.OptionalParameters("Poke!", active: false));
// command Poke!, number 0, active False
```

Mixing optional and required parameters

```
0 references
public string OptionalParameters(string command = "Run!", double number = 0.0, bool active = true, int count)

// Error CS1737 Optional parameters must appear after all required parameters

0 references
public string OptionalParameters(int count, string command = "Run!", double number = 0.0, bool active = true)

0 references
WriteLine(bob.OptionalParameters(3));
0 references
WriteLine(bob.OptionalParameters(3, "Jump!", 98.5));
0 references
WriteLine(bob.OptionalParameters(3, number: 52.7, command: "Hide!"));
0 references
WriteLine(bob.OptionalParameters(3, "Poke!", active: false));
```

Controlling how parameters are passed

A parameter can be passed in one of several ways:

- By **value** (default): **in-only**
  - value can be changed; this only affects the parameter in the method
- As an **out parameter**: **out-only**
  - out parameters cannot have a default value assigned in their declaration
  - out parameters cannot be left uninitialized
  - must be set inside the method; otherwise, the compiler will give an error
- By **reference** as a **ref** parameter: **in-and-out**
  - ref parameters cannot have default values
  - they do not need to be set inside the method
- As an **in parameter**: **reference parameter** that is **read-only**
  - in parameters cannot have their value changed
  - the compiler will show an error if you try

Controlling how parameters are passed

```
0 references
public void PassingParameters(int w, in int x, ref int y, out int z) {
    // out parameters cannot have a default and they
    // must be initialized inside the method.
    z = 100;
    // Increment each parameter except the read-only x.
    w++;
    // x++; // Gives a compiler error!
    y++;
    z++;
    WriteLine($"In the method: w={w}, x={x}, y={y}, z={z}");
}
0 references
WriteLine($"Before: a={a}, b={b}, c={c}, d={d}");
```

Controlling how parameters are passed

- When passing a variable as a parameter by default, its current value gets passed, not the variable itself
  - Therefore, parameter **w** has a copy of the value of variable **a**
  - Variable **a** retains its original value even after parameter **w** is incremented
- When passing a variable as an **in** parameter, a reference to the variable gets passed into the method
  - Therefore, parameter **x** is a reference to variable **b**
  - If variable **b** gets incremented by some other process while the method is executing, then parameter **x** would show that

Controlling how parameters are passed

- When passing a variable as a **ref** parameter, a reference to the variable gets passed into the method
  - Therefore, parameter **y** is a reference to variable **c**
  - Variable **c** gets incremented when parameter **y** gets incremented
- When passing a variable as an out parameter, a reference to the variable gets passed into the method
  - Therefore, parameter **z** is a reference to variable **d**
  - The value of variable **d** gets replaced by whatever code executes inside the method

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
  - complex type, such as Person
  - collection type, such as List<Person>
- Imagine that we want to define a method named GetTheData that needs to return 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

Combining multiple returned values using tuples

- Defining a class just to combine two values together is unnecessary
- Modern versions of C# supports **tuples**
- Tuples are an efficient way to combine two or more values into a single unit
- Tuples have been a part of some languages such as F# since their first version
- .NET added support for them using the **System.Tuple** type

```
public class TextAndNumber {
    public string Text;
    public int Number;
}

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

Language support for tuples

- C# 7.0 added language syntax support for tuples using the parentheses characters ()
- .NET added a new **System.ValueTuple** type that is more efficient in some common scenarios than the old .NET 4.0 **System.Tuple** type
- The C# tuple syntax uses the more efficient one

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

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

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

Naming the fields of a tuple

- To access the fields of a tuple, the default names are Item1, Item2, …
- You can explicitly specify the field names

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

var fruitNamed = bob.GetNamedFruit();

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

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

Controlling access with properties and indexers

- A fundamental difference between a field and a property is that a field provides a memory address to data
- You could pass that memory address to an external component, like a Windows API C-style function call, and it could then modify the data
- A property does not provide a memory address to its data, which provides more control
- All you can do is ask the property to get or set the data
- The property then executes statements and can decide how to respond, including refusing the request!

Defining read-only properties

```
// 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 body syntax
public string Greeting => $"{Name} says 'Hello!'";

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

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

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

Defining settable properties

```
// auto property - read and write
public string? FavoriteIceCream { get; set; }
// private field to store the property value
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.");
        }
    }
}
```

Defining settable properties

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

string color = "Red";
try {
    sam.FavoritePrimaryColor = color;
    WriteLine($"Sam's favorite primary color is
        {sam.FavoritePrimaryColor}.");
}
catch (Exception ex) {
    WriteLine("Tried to set {0} to '{1}': {2}",
        nameof(sam.FavoritePrimaryColor), color, ex.Message);
}
```

Requiring properties to be set during instantiation

- C# 11 and .NET 7.0 introduces the **required** modifier
- If you use it on a property, the compiler will ensure that you set the property to a value when you instantiate it

```
// Introduced with C# 11 and .NET 7.0
public class Book {
    public required string? Isbn { get; set; }
    public required string? Title { get; set; }
    public string? Author { get; set; }
    public int PageCount { get; set; }
}

Book book = new() {
    Isbn = "978-1803237800",
    Title = "C# 11 and .NET 7 - Modern Cross-Platform Development Fundamentals"
};
```

Requiring properties to be set during instantiation

```
Book book = new();
```

```
C:\cs11dotnet7-old\Chapter05\PeopleApp\Program.cs(164,13): error CS9035:
Required member 'Book.Isbn' must be set in the object initializer or
attribute constructor. [C:\cs11dotnet7-old\Chapter05\PeopleApp\PeopleApp.
csproj]
C:\cs11dotnet7-old\Chapter05\PeopleApp\Program.cs(164,13): error CS9035:
Required member 'Book.Title' must be set in the object initializer or
attribute constructor. [C:\cs11dotnet7-old\Chapter05\PeopleApp\PeopleApp.
csproj]

    0 Warning(s)
    2 Error(s)
```

Defining indexers

- Indexers allow the calling code to use the array syntax to access a property
- E.g., string type defines an **indexer** so that the calling code can access individual characters in the string

```
// indexers
public Person this[int index] {
    get {
        // pass on to the List<T> indexer
        return Children[index];
    }
    set {
        Children[index] = value;
    }
}
```

```
// overloaded indexers
public Person this[string name] {
    get {

        return Children.Find(p => p.Name == name);
    }
    set {
        Person found = Children.Find(p => p.Name = name);
        if (found is not null) found = value;
    }
}
```

```
sam.Children.Add(new() { Name = "Charlie", DateOfBirth = new(2010, 3, 18) });
sam.Children.Add(new() { Name = "Ella", DateOfBirth = new(2020, 12, 24) });

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}");
```

Init-only properties

- You have used object initialization syntax to instantiate objects and set initial properties
- Those properties can also be changed after instantiation
- Sometimes, you want to treat properties like **readonly** fields so that they can be set during instantiation but not after
- In other words, they are immutable
- The **init** keyword enables this
- It can be used in place of the set keyword in a property definition

Init-only properties

- You have used object initialization syntax to instantiate objects and set initial properties
- Those properties can also be changed after instantiation
- Sometimes, you want to treat properties like **readonly** fields so that they can be set during instantiation but not after
- In other words, they are immutable
- The **init** keyword enables this
- It can be used in place of the set keyword in a property definition

Init-only properties

```
1 reference
public class ImmutablePerson {
    1 reference
    public string? FirstName { get; init; }
    1 reference
    public string? LastName { get; init; }
}
0 references
ImmutablePerson jeff = new() {
    FirstName = "Jeff",
    LastName = "Winger"
};
jeff.FirstName = "Geoff";
```

```
C:\cs12dotnet8\Chapter05\PeopleApp\Program.cs(404,1): 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]
```

Defining record types

- Init-only properties provide some immutability to C#
- You can take the concept further by using **record types**
- These are defined by using the record keyword instead of the class keyword
- That can make the whole object immutable, and it acts like a value when compared
- Immutable records should not have any state (properties and fields) that change after instantiation
- Instead, the idea is that you create new records from existing ones
- The new record has the changed state
- This is called non-destructive mutation

Defining record types

```
2 references
public record ImmutableVehicle {
    1 reference
    public int Wheels { get; init; }
    1 reference
    public string? Color { get; init; }
    1 reference
    public string? Brand { get; init; }
}
0 references
ImmutableVehicle car = new() {
    Brand = "Mazda MX-5 RF",
    Color = "Soul Red Crystal Metallic",
    Wheels = 4
};
0 references
ImmutableVehicle repaintedCar = car
    with { Color = "Polymetal Grey Metallic" };
0 references
WriteLine($"Original car color was {car.Color}.");
//  Original car color was Soul Red Crystal Metallic.
0 references
WriteLine($"New car color is {repaintedCar.Color}.");
//  New car color is Polymetal Grey Metallic.
```