CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 6 – Part 3

Inheriting from classes

- The Person type (class) created previously derived (inherited) from object (System.Object)
- Modify its contents to define a class named Employee that derives from Person

```
27 references
public class Person : object { … }
1 reference
public class Employee : Person { … }    // inherits all Person class members
```

- add statements to create an instance of the Employee class

```
Employee john = new() {
    Name = "John Jones",
    Born = new(year: 1990, month: 7, day: 28,
        hour: 0, minute: 0, second: 0, offset: TimeSpan.Zero)
};
john.WriteToConsole();
```

Extending classes to add functionality

- Add statements to define two properties for an employee code and the date they were hired

```
2 references
public class Employee : Person {
        1 reference
        public string? EmployeeCode { get; set; }
        2 references
        public DateOnly HireDate { get; set; }
}
```

- Add statements to set John's employee code and hire date

```
john.EmployeeCode = "JJ001";
john.HireDate = new(year: 2014, month: 11, day: 23);
WriteLine($"{john.Name} was hired on {john.HireDate:dd/MM/yy}");
```

Hiding (Replacing) members

- Add statements to redefine the WriteToConsole method

```
3 references
public class Employee : Person {
    1 reference
    public string? EmployeeCode { get; set; }
    3 references
    public DateOnly HireDate { get; set; }
    1 reference
    public new void WriteToConsole() {  // deliberately replacing the old method
        WriteLine(format: "{0} was born on {1:dd/MM/yy} and hired on {2:dd/MM/yy}.",
            arg0: Name,
            arg1: Born,
            arg2: HireDate);
    }
}
```

Understanding the this and base keywords

- **this**:
  - Represents the current object instance
  - E.g., in the Person class instance members, you could use the expression this.Born to access the Born field of the current object instance
  - You rarely need to use it, since the expression Born would also work
  - It is only when there is a local variable also named Born that you would need to use this.Born, to explicitly say you are referring to the field, not the local variable
- **base**:
  - Represents the base class that the current object inherits from
  - E.g., anywhere in the Person class, you could use the expression base.ToString() to call the base class implementation of that method

Overriding members

- Rather than hiding a method, it is usually better to **override** it
- You can only override if the base class chooses to allow overriding, by applying the **virtual** keyword to any methods that should allow overriding
- add a statement to write the value of the john variable to the console using its string representation

```
public override string ToString() {                    // base class must specify virtual
    return $"{Name} is a {base.ToString()}";
}
```

- The **base** keyword allows a **subclass** to access members of its **superclass**
  - The **base class** that it inherits or derives from

Inheriting from abstract classes

- Previously you learned about interfaces that can define a set of members that a type must have to meet a basic level of functionality
- Their main limitation is that until C# 8.0 they could not provide any implementation of their own
- You can use **abstract classes** as a sort of halfway house between a **pure interface** and a **fully implemented class**
- When a class is marked as **abstract**, it **cannot be instantiated** because you are indicating that the class is not complete
- It needs more implementation before it can be instantiated

Inheriting from abstract classes

- Let's compare the two types of interface and two types of class

```
public interface INoImplementation {    // C# 1.0 and later
    void Alpha();                       // must be implemented by derived type
}

public interface ISomeImplementation {  // C# 8.0 and later
    void Alpha();                       // must be implemented by derived type
    void Beta() {
        ...
    }
}
```

Inheriting from abstract classes

- Let's compare the two types of interface and two types of class

```
public abstract class PartiallyImplemented { // C# 1.0 and later
    public abstract void Gamma();   // must be implemented by derived type
    public virtual void Delta() {   // can be overridden
        // implementation
    }
}

public class FullyImplemented : PartiallyImplemented, ISomeImplementation {
    public void Alpha() {
        // implementation
    }
    public override void Gamma() {
        // implementation
    }
}
```

Inheriting from abstract classes

- Let's compare the two types of interface and two types of class

```
// you can only instantiate the fully implemented class
FullyImplemented a = new();

// all the other types give compile errors
PartiallyImplemented b = new();      // compile error!
ISomeImplementation c = new();       // compile error!
INoImplementation d = new();         // compile error!
```

Choosing between an interface and an abstract class

- Which should you pick?
- Now that an interface can have default implementations for its members, is the abstract keyword for a class obsolete?
- Well, let’s think about a real example. Stream is an abstract class
  - Would or could the .NET team use an interface for that today?
- Every member of an interface must be public
  - An abstract class has more flexibility in its members’ access modifiers
- Another advantage of an abstract class over an interface is that serialization often does not work for an interface
- So, no, we still need to be able to define abstract classes

Preventing inheritance and overriding

- You can prevent another developer from inheriting from your class by applying the **sealed** keyword to its definition

```
public sealed class ScroogeMcDuck { }
```

- An example of sealed in .NET is the string class
- Microsoft has implemented some extreme optimizations inside the string class that could be negatively affected by your inheritance

Preventing inheritance and overriding

- You can prevent someone from further overriding a virtual method in your class by applying the **sealed** keyword to the method

```
public class Singer {
    // virtual allows this method to be overridden
    public virtual void Sing() {
        WriteLine("Singing...");
    }
}

public class LadyGaga : Singer {
    // sealed prevents overriding the method in subclasses
    public sealed override void Sing() {
        WriteLine("Singing with style...");
    }
}
```

Understanding Polymorphism

- You have now seen two ways to change the behavior of an inherited method
  - We can hide it using the **new** keyword
    - **non-polymorphic inheritance**
  - We can override it using the **virtual/override** keywords
    - **polymorphic inheritance**
- Both ways can access members of the base or superclass by using the base keyword
- What is the difference?
- It all depends on the type of variable holding a reference to the object
  - E.g., a variable of the Person type can hold
    - a reference to a Person class
    - any type that derives from Person

Understanding Polymorphism

- When a method is hidden with **new**, the compiler is not smart enough to know that the object is an Employee
  - It calls the WriteToConsole method in Person
- When a method is overridden with **virtual** and **override**, the compiler is smart enough to know that although the variable is declared as a Person class, the object itself is an Employee class
  - Employee implementation of ToString is called

Variable type	**Member modifier**	**Method executed**	**In class**
`Person`		`WriteToConsole`	`Person`
`Employee`	`new`	`WriteToConsole`	`Employee`
`Person`	`virtual`	`ToString`	`Employee`
`Employee`	`override`	`ToString`	`Employee`

Abstract/Concrete Inheritance/Polymorphism Example

```
namespace Gamradt3;

5 references
public abstract class Base
{
    3 references
    public int One { get; set; }
    2 references
    public Base(int One = 5) {
        this.One = One;
    }
    1 reference
    public Base(Base instance) {
        One = instance.One;
    }
    public override string? ToString()
    {
        return base.ToString()
        // + string describing Base class data
        ;
    }
}
```

```
namespace Gamradt3;

1 reference
public class Derived : Base
{
    4 references
    public double Two { get; set; }
    0 references
    public Derived() : base() {
        this.Two = 7.5;
    }
    0 references
    public Derived(int One, double Two = 7.5) : base(One) {
        this.Two = Two;
    }
    0 references
    public Derived(Derived instance) : base(instance) {
        Two = instance.Two;
    }
    public override string? ToString()
    {
        return base.ToString()
        // + string describing Derived class data
        ;
    }
}
```

Casting with inheritance hierarchies

- Casting between types is subtly different from converting between types
- Casting is between similar types
  - between a 16-bit integer and a 32-bit integer
  - between a superclass and one of its subclasses
- Converting is between dissimilar types, such as between text and a number

Implicit casting

- In the previous example, you saw how an instance of a derived type can be stored in a variable of its base type (or its base's base type, ...)
- When we do this, it is called **implicit casting**

Explicit casting

- Going the other way is an explicit cast, and you must use parentheses around the type you want to cast into as a prefix to do it

Employee explicitAlice = aliceInPerson;

- Change the statement to prefix the assigned variable named with a cast to the Employee type

Employee explicitAlice = **(Employee)**aliceInPerson;

Avoiding casting exceptions

- The compiler is now happy
- Because aliceInPerson might be a different derived type, like Student instead of Employee, we need to be careful
- In a real application with more complex code, the current value of this variable could have been set to a Student instance, and then this statement would throw an **InvalidCastException** error

Using is to type check

- We can handle this by writing a try statement, but there is a better way
- We can check the type of an object using the **is** keyword

```
if (aliceInPerson is Employee) {
    WriteLine($"{nameof(aliceInPerson)} IS an Employee");
    Employee explicitAlice = (Employee)aliceInPerson;
    // safely do something with explicitAlice
}
```

Using is to type check

- You can simplify the code further using a declaration pattern and this will avoid needing to perform an explicit cast

```
if (aliceInPerson is Employee explicitAlice) {
    WriteLine($"{nameof(aliceInPerson)} IS an Employee");
    // safely do something with explicitAlice
}
```

- What if you want to execute a block of statements when Alice is not an employee?

```
if ( ! (aliceInPerson is Employee))          // older C# versions
if (aliceInPerson is not Employee)           // C# 9.0 and later
```

Using as to cast a type

- You can use the **as** keyword to cast
- The **as** keyword returns null if the type cannot be cast

```
// could be null
Employee? aliceAsEmployee = aliceInPerson as Employee;
if (aliceAsEmployee is not null) {
    WriteLine($"{nameof(aliceInPerson)} as an Employee.");
    // Safely do something with aliceAsEmployee
}
```

Inheriting and extending .NET types

- .NET has prebuilt class libraries containing hundreds of thousands of types
- Rather than creating your own completely new types, you can often get a head start by deriving from one of Microsoft's types to inherit some or all of its behavior and then overriding or extending it

Inheriting exceptions

- As an example of inheritance, we will derive a new type of exception

```
3 references
public class PersonException : Exception {
    0 references
    public PersonException() : base() { }
    0 references
    public PersonException(string message) : base(message) { }
    0 references
    public PersonException(string message, Exception innerException)
        : base(message, innerException) { }
}
```

- Unlike ordinary methods, **constructors are not inherited**, so we must explicitly declare and explicitly call the base constructor implementations in System.Exception to make them available to programmers who might want to use those constructors with our custom exception

Inheriting exceptions

- Add statements to define a method that throws an exception if a date/ time parameter is earlier than a person's date of birth

```
public void TimeTravel(DateTime when) {
    if (when <= Born) {
        throw new PersonException("If you travel back in time to a date "
            + "earlier than your own birth, then the universe will explode!");
    }
    else {
        WriteLine($"Welcome to {when:yyyy}!");
    }
}
```

Inheriting exceptions

- Add statements to test what happens when employee John Jones tries to time travel too far back

```
try {
    john.TimeTravel(when: new(1999, 12, 31));
    john.TimeTravel(when: new(1950, 12, 25));
} catch (PersonException ex) {
    WriteLine(ex.Message);
}
```

Extending types when you can't inherit

- Previously, we saw how the **sealed** modifier can be used to prevent inheritance
- Microsoft has applied the sealed keyword to the System.String class so that no one can inherit and potentially break the behavior of strings
- Can we still add new methods to strings?
- Yes …
- If we use a language feature named **extension methods**, which was introduced with C# 3.0

Using static methods to reuse functionality

- We can create **static methods** to reuse functionality
  - E.g., Ability to validate that a string contains an email address

```
// To use Regex. namespace Packt.Shared;
using System.Text.RegularExpressions;
0 references
public class StringExtensions {
    0 references
    public static bool IsValidEmail(string input) {
        // Use a simple regular expression to check
        // that the input string is a valid email.
        return Regex.IsMatch(
            input,
            @"[a-zA-Z0-9\.-_]+@[a-zA-Z0-9\.-_]+"
        );
    }
}
```

Using static methods to reuse functionality

- The static IsValidEmail method uses the Regex type to check for matches against a simple email pattern that looks for valid characters before and after the @ symbol

```
string email1 = "pamela@test.com";  // valid       True
string email2 = "ian&test.com";     // not valid   False
WriteLine("{0} is a valid e-mail address: {1}",
    arg0: email1,
    arg1: StringExtensions.IsValidEmail(email1));
WriteLine("{0} is a valid e-mail address: {1}",
    arg0: email2,
    arg1: StringExtensions.IsValidEmail(email2));
```

Using extension methods to reuse functionality

- It is easy to make static methods into extension methods

```
using System.Text.RegularExpressions;

public static class StringExtensions {
    public static bool IsValidEmail(this string input) {
        // use simple regular expression to check that the input string is a valid email
        return Regex.IsMatch(input, @"[a-zA-Z0-9\.-_]+@[a-zA-Z0-9\.-_]+");
    }
}
```

- These two changes tell the compiler that it should treat the method as one that extends the string type

Categories of custom types and their capabilities

Type	Instantiation	Inheritance	Equality	Memory
`class`	Yes	Single	Reference	Heap
`sealed class`	Yes	None	Reference	Heap
`abstract class`	No	Single	Reference	Heap
`record` or `record class`	Yes	Single	Value	Heap
`struct` or `record struct`	Yes	None	Value	Stack
`interface`	No	Multiple	Reference	Heap

Categories of custom types and their capabilities

- Now let's highlight what is different about the more specialized types of classes:
  - A sealed class does not support inheritance
  - An abstract class does not allow instantiation with new
  - A record class uses value equality instead of reference equality
- We can do the same for other types compared to a "normal" class:
  - A struct or record struct does not support inheritance, it uses value equality instead of reference equality, and its state is stored in stack memory
  - An interface does not allow instantiation with new and supports multiple inheritance