Answer on Question#41052- Programming, C#

1. Reuse the functions of the book Tickets class in a new class, named book Ticket, to add the feature of e-ticket booking create a c# programming

Solution.

Object-Oriented Programming (C#)

All managed languages in the .NET Framework, such as Visual Basic and C#, provide full support for object-oriented programming including encapsulation, inheritance, and polymorphism.
Encapsulation means that a group of related properties, methods, and other members are treated as a single unit or object.
Inheritance describes the ability to create new classes based on an existing class.
Polymorphism means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.
This section describes the following concepts:

Classes and Objects

The terms *class* and *object* are sometimes used interchangeably, but in fact, classes describe the *type* of objects, while objects are usable*instances* of classes. So, the act of creating an object is called *instantiation*. Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.
To define a class:

```
class SampleClass
{
}
```

Both Visual Basic and C# also provide a light version of classes called *structures* that are useful when you need to create large array of objects and do not want to consume too much memory for that.
To define a structure:

```
struct SampleStruct
{
}
```

Class Members

Each class can have different *class members* that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects.

Properties and Fields

Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.
To define a field:

```
Class SampleClass
```

```
{
    public string sampleField;
}
```

Properties have get and set procedures, which provide more control on how values are set or returned. Both C# and Visual Basic allow you either to create a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures.
To define an auto-implemented property:

```
class SampleClass
{
    public int SampleProperty { get; set; }
}
```

If you need to perform some additional operations for reading and writing the property value, define a field for storing the property value and provide the basic logic for storing and retrieving it:

```
class SampleClass
{
    private int _sample;
    public int Sample
    {
        // Return the value stored in a field.
        get { return _sample; }
        // Store the value in the field.
        set { _sample = value; }
    }
}
```

Most properties have methods or procedures to both set and get the property value. However, you can create read-only or write-only properties to restrict them from being modified or read. In Visual Basic you can use **ReadOnly** and **WriteOnly** keywords. In C#, you can omit the **get** or **set** property method. However, in both Visual Basic and C#, auto-implemented properties cannot be read-only or write-only.

Methods

A *method* is an action that an object can perform.
To define a method of a class:

```
class SampleClass
{
    public int sampleMethod(string sampleParam)
    {
        // Insert code here
    }
}
```

A class can have several implementations, or *overloads*, of the same method that differ in the number of parameters or parameter types.

To overload a method:

```
public int sampleMethod(string sampleParam) {};
public int sampleMethod(int sampleParam) {}
```

In most cases you declare a method within a class definition. However, both Visual Basic and C# also support *extension methods* that allow you to add methods to an existing class outside the actual definition of the class.

Constructors

Constructors are class methods that are executed automatically when an object of a given type is created. Constructors usually initialize the data members of the new object. A constructor can run only once when a class is created. Furthermore, the code in the constructor always runs before any other code in a class. However, you can create multiple constructor overloads in the same way as for any other method.
To define a constructor for a class:

```
public class SampleClass
{
    public SampleClass()
    {
        // Add code here
    }
}
```

Destructors

Destructors are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need destructors to clean up any unmanaged resources that your application creates. There can be only one destructor for a class.
For more information about destructors and garbage collection in the .NET Framework, see Garbage Collection.

Events

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the *publisher* and the classes that receive (or handle) the event are called *subscribers*. For more information about events, how they are raised and handled, see

Nested Classes

A class defined within another class is called *nested*. By default, the nested class is private.

```
class Container
{
    class Nested
    {
        // Add code here.
    }
}
```

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:
`Container.Nested nestedInstance = new Container.Nested()`

Access Modifiers and Access Levels

All classes and class members can specify what access level they provide to other classes by using *access modifiers*.
The following access modifiers are available:

C# Modifier	Definition

public	The type or member can be accessed by any other code in the same assembly or another assembly that references it.
private	The type or member can only be accessed by code in the same class.
protected	The type or member can only be accessed by code in the same class or in a derived class.
internal	The type or member can be accessed by any code in the same assembly, but not from another assembly.
protected internal	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

Instantiating Classes

To create an object, you need to instantiate a class, or create a class instance.

```
SampleClass sampleObject = new SampleClass();
```

After instantiating a class, you can assign values to the instance's properties and fields and invoke class methods.

```
// Set a property value.
sampleObject.sampleProperty = "Sample String";
// Call a method.
sampleObject.sampleMethod();
```

To assign values to properties during the class instantiation process, use object initializers:

```
// Set a property value.
SampleClass sampleObject = new SampleClass
    { FirstProperty = "A", SecondProperty = "B" };
```

Static (Shared) Classes and Members

A static (shared in Visual Basic) member of the class is a property, procedure, or field that is shared by all instances of a class.
To define a static (shared) member:

```
static class SampleClass
{
    public static string SampleString = "Sample String";
}
```

To access the static (shared) member, use the name of the class without creating an object of this class:

```
Console.WriteLine(SampleClass.SampleString);
```

Static (shared) classes in C# and modules in Visual Basic have static (shared) members only and cannot be instantiated. Static (shared) members also cannot access non-static (non-shared) properties, fields or methods

Anonymous Types

Anonymous types enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. The class has no usable name and contains the properties you specify in declaring the object.

To create an instance of an anonymous type:

```
// sampleObject is an instance of a simple anonymous type.
var sampleObject =
    new { FirstProperty = "A", SecondProperty = "B" };
```

Inheritance

Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in both C# and Visual Basic implicitly inherit from the Object class that supports .NET class hierarchy and provides low-level services to all classes.

```
class DerivedClass:BaseClass{}
```

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.
To specify that a class cannot be used as a base class:

```
public sealed class A { }
```

To specify that a class can be used as a base class only and cannot be instantiated:

```
public abstract class B { }
```

Overriding Members

By default, a derived class inherits all members from its base class. If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.
The following modifiers are used to control how properties and methods are overridden:

C# Modifier	Definition
virtual (C# Reference)	Allows a class member to be overridden in a derived class.
override (C# Reference)	Overrides a virtual (overridable) member defined in the base class.
Not supported	Prevents a member from being overridden in an inheriting class.
abstract (C# Reference)	Requires that a class member to be overridden in the derived class.
new Modifier (C# Reference)	Hides a member inherited from a base class

Interfaces

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.
To define an interface:

```
interface ISampleInterface
{
    void doSomething();
}
```

To implement an interface in a class:

```
class SampleClass : ISampleInterface
{
    void ISampleInterface.doSomething()
    {
        // Method implementation.
    }
}
```

Generics

Classes, structures, interfaces and methods in the .NET Framework can *type parameters* that define types of objects that they can store or use. The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.
To define a generic class:

```
Public class SampleGeneric<T>
{
    public T Field;
}
```

To create an instance of a generic class:

```
SampleGeneric<string> sampleObject = new SampleGeneric<string>();
sampleObject.Field = "Sample string";
```

Delegates

A *delegate* is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.
To create a delegate:

```
public delegate void SampleDelegate(string str);
```

To create a reference to a method that matches the signature specified by the delegate:

```
class SampleClass
{
    // Method that matches the SampleDelegate signature.
    public static void sampleMethod(string message)
    {
        // Add code here.
    }
    // Method that instantiates the delegate.
    void SampleDelegate()
    {
        SampleDelegate sd = sampleMethod;
        sd("Sample string");
    }
}
```