OBJECT ORIENTED PROGRAMMING

Unit 2

CONTENTS

- Understanding the C# Class Type
- Reviewing the Pillars of OOP
- C#'s Encapsulation Services
- C#'s Inheritance Support
- Programming for Containment/Delegation
- C #'s Polymorphic Support, C# Casting rules
- Understanding C# Partial types
- Documenting C# Source Code via XML
- Understanding Object Lifetime Classes

CONTENTS

- Objects and References
- The basics of Object Lifetime
- System.GC type
- Building Finalizable Objects
- Building Disposable Objects

UNDERSTANDING THE C# CLASS TYPE

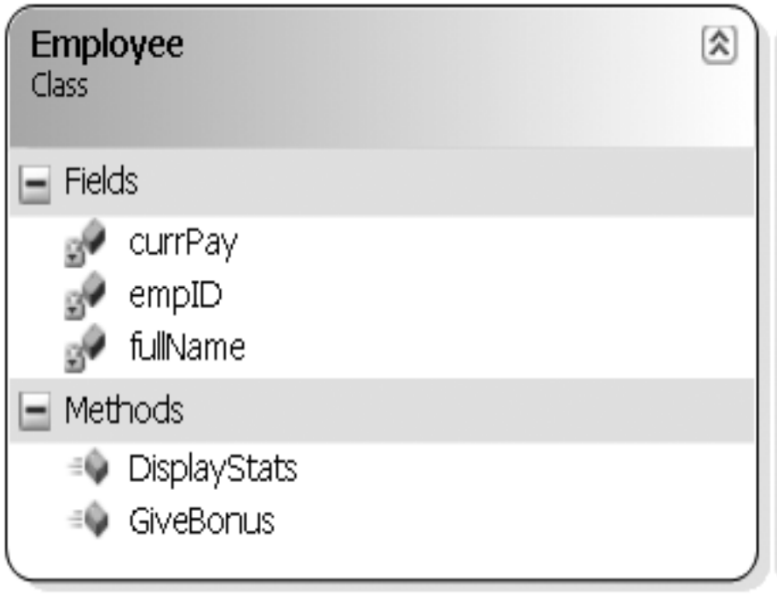
- Formally, a class is nothing more than a custom user-defined type (UDT) that is composed of field data (sometimes termed *member variables*) and functions (often called *methods*) that act on this data.
- The set of field data collectively represents the "state" of a class instance.
- The power of object-oriented languages is by grouping data and functionality in a single UDT, to model your software types into real-world entities.

UNDERSTANDING THE C# CLASS TYPE

- For example, if you wish to create a generic employee for a payroll system, you may wish to build a class that maintains the name, current pay, and employee ID for each worker.
- In addition, the Employee class defines a method, named GiveBonus(), which increases an individual's current pay by some amount, and another, named DisplayStats(), which prints out the state data for this individual

UNDERSTANDING THE C# CLASS TYPE

- Figure 2-1 illustrates the Employee class type.



- C# classes can define any number of *constructors*.
- These special class methods provide a simple way for a user to create an instance of a class.
- Every class is initially provided with a *default constructor*, which never takes arguments.
- In addition to the default constructor, you are also free to define as many constructors which can take parameters *(parameterized constructors)*.

UNDERSTANDING THE C# CLASS TYPE

- To create an instance of a class we need to create objects.
- Objects in C# are created using the new keyword.
- The syntax for creating an instance of a class object is as follows:

```
ClassName object-name = new ClassName();
```

UNDERSTANDING THE C# CLASS TYPE

- Like C++ and Java, if you choose to define custom constructors in a class definition, the default constructor is *silently removed*.
- Therefore, if you wish to create an instance of Employee class then you can create this as follows:

```
// Calls the default constructor.
Employee e = new Employee();
```

- You must explicitly redefine the default constructor for your class to execute the above code.

UNDERSTANDING THE C# CLASS TYPE

- If you do not, you will receive a compiler error when creating an instance of your class type using the default constructor.
- To call the parameterized constructor of the Employee class, we need to use the following code:

```
Employee e = new Employee ("Joe", 80, 30000);
```

- **Program :ch02pg01.cs**

METHOD OVERLOADING

- Like other object-oriented languages, C# allows a type to *overload* various methods.
- Simply put, when a class has a set of identically named methods that differ by the number (or type) of parameters, the method in is said to be *overloaded*.
- For example:
  - Assume you have a class named Triangle that supports an overloaded Draw() method.
  - By doing so, you allow the object user to render the image using various input parameters:

METHOD OVERLOADING

```
public class Triangle
{
// The overloaded Draw() method.
public void Draw(int x, int y, int height, int
width) {...}

public void Draw(float x, float y, float
height, float width) {...}

public void Draw(Point upperLeft, Point
bottomRight) {...}

public void Draw(Rect r) {...}
}
```

Program: ch01pg02.cs

CONSTRUCTOR OVERLOADING

- Not only methods but you can also overload constructors in a class. This concept is called as constructor overloading. For example:

```
public class Employee
{
  ...
  // Overloaded constructors.
  public Employee(){ }
  public Employee(string fullName, int
  empID, float currPay){...}
  ...
}
```

Program:ch02pg03.cs

SELF-REFERENCE IN C# USING *THIS*

- C# '*this*' keyword is used when you wish to explicitly reference the fields and members of the *current object*.
- The main use of '*this*' in your class members is to avoid clashes between the parameter names and names of member variables. For example:

```
public Employee(string name, int id, float pay)
{
this.fullName = name;
this.empID = id;
this.currPay = pay;
}
```

FORWARDING CONSTRUCTOR CALLS USING THIS

- Another use of *this* keyword is to force one constructor to call another in order to avoid redundant member initialization logic.
- The general form is shown here:

constructor-name(*parameter-list1*) : this(*parameter-list2*) {
// ... body of constructor, which may be empty
}

REVIEWING THE PILLARS OF OOP

- *Encapsulation*:
  - The first pillar of OOP is called *encapsulation*.
  - This trait boils down to the language's ability to hide unnecessary implementation details from the object user.
- *Inheritance*:
  - The next pillar of OOP, inheritance, boils down to the language's ability to allow you to build new class definitions based on existing class definitions.

REVIEWING THE PILLARS OF OOP

- *Polymorphism*:
  - The final pillar of OOP is *polymorphism* which is the ability to treat related objects the same way.
  - This concept of object-oriented language allows a base class to define a set of members (formally termed the *polymorphic interface*) to all descendents.
  - A class type's polymorphic interface is constructed using any number of *virtual* or *abstract* members.

REVIEWING THE PILLARS OF OOP

- Polymorphism(Contd)
  - In a nutshell, a virtual member *may* be changed (or more formally speaking, *overridden*) by a derived class, whereas an abstract method *must* be overriden by a derived type.
  - When derived types override the members defined by a base class, they are essentially redefining how they respond to the same request.

C#'S ENCAPSULATION SERVICES

- The concept of encapsulation revolves around the notion that an object's field data should not be directly accessible from the public interface.
- In C#, encapsulation is enforced at the syntactic level using the public, private, protected, and protected internal keywords.
- Encapsulation provides a way to preserve the integrity of state data.

C#'S ENCAPSULATION SERVICES

- Rather than defining public fields (which can easily foster data corruption), you should get in the habit of defining *private data fields*, which are indirectly manipulated by the caller using one of two main techniques:
  - Define a pair of traditional accessor and mutator methods.
  - Define a named property.

C#'S ENCAPSULATION SERVICES

- **Enforcing Encapsulation Using Traditional Accessors and Mutators**
  - If you want the outside world to interact with your private *fullName* data field, tradition dictates defining an *accessor* (get method) and *mutator* (set method).
  - For example:

```
// Traditional accessor and mutator for a
point of private data.
public class Employee
{
private string fullName;
// Accessor.
public string GetFullName() { return
fullName; }
```

C#'S ENCAPSULATION SERVICES

```
// Mutator.
public void SetFullName(string n)
{
// Remove any illegal characters (!,
@, #, $, %),
// check maximum length (or case
rules) before making assignment.
fullName = n;
}
```

CLASS PROPERTIES

- In contrast to traditional accessor and mutator methods, .NET languages prefer to enforce encapsulation using *properties*, which simulate publicly accessible points of data.
- Rather than requiring the user to call two different methods to get and set the state data, the user will call the property like a public field.

CLASS PROPERTIES

- The general form of a property is shown here:

```
type name {
   get {
   // get accessor code
   }
   set {
   // set accessor code
   }
}
```

- Here, *type* specifies the data type of the property, such as **int**, and *name* is the name of the property.

CLASS PROPERTIES

- Once the property has been defined, any use of *name* results in a call to its appropriate accessor.
- The **set** accessor automatically receives a parameter called **value** that contains the value being assigned to the property.
- **Program ch02pg05.cs:** *Here is a simple example that defines a property called MyProp, which is used to access the field prop. In this case, the property allows only positive values to be assigned.*

CONTROLLING VISIBILITY LEVELS OF PROPERTY

- By default, the **set** and **get** accessors have the same accessibility as property of which they are a part.
- For example, if the property is declared **public**, then by default the **get** and **set** accessors are also public.
- It is possible, however, to give **set** or **get** its own access modifier, such as **private or protected**.
- In all cases, the access modifier for an accessor must be more restrictive then the access specification of its property.

CONTROLLING VISIBILITY LEVELS OF PROPERTY

- **Program ch02pg06.cs:**
- Program to demonstrate the use an access modifier with an accessor.
- Here is a property called **MyProp** that has its **set** accessor specified as **private**.

READ-ONLY AND WRITE-ONLY PROPERTIES

- Properties can be defined as read-only, write-only or read-write property depending upon its underlying field to be read or written.
- To create a read-only property, define only a get accessor.
- To define a write-only property, define only a set accessor.
- **Program ch02pg07.cs**: This program demonstrates read-only property. Here we have a *Customer* class which has two read-only properties, *ID* and *Name*.

READ-ONLY AND WRITE-ONLY PROPERTIES

- **Program ch02pg08.cs**: This program demonstrates the use of write-only property. Here we have a *Customer* class which has two write-only properties, *ID* and *Name*. This time, the *get* accessor is removed from the *ID* and *Name* properties of the *Customer* class. The *set* accessors have been added, assigning *value* to the backing store fields, *m_id* and *m_name*.

- **Program ch02pg09.cs :** This program shows a Person class that has two read-write properties: Name (string) and Age (int). Both properties provide **get** and **set** accessors, so they are considered read/write properties.

READ-ONLY AND WRITE-ONLY PROPERTIES

- When a property declaration includes a static modifier, the property is said to be a static property.
- A static property is not associated with a specific instance, and it is a compile-time error to refer to this in the accessors of a static property.
- **Program ch02pg10.cs:** This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments numberOfEmployees by 1, and displays the Employee name and number.

C#'S INHERITANCE SUPPORT

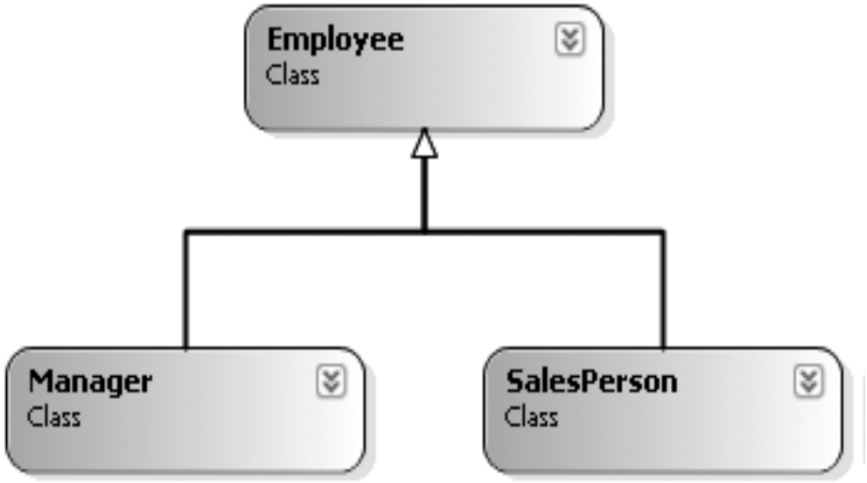
- Inheritance is the aspect of OOP that facilitates code reuse.
- In C#, a class that is inherited is called a *base class*.
- The class that does the inheriting is called a *derived class*.
- Therefore, a derived class is a specialized version of a base class.
- It inherits all of the variables, methods, properties, and indexers defined by the base class and add its own unique elements.
- In C#, extending a class is accomplished using the colon operator (:) on the class definition.

C#'S INHERITANCE SUPPORT

- Inheritance comes in two flavors:
  - Classical inheritance (the "is-a" relationship) and
  - The containment/ delegation model (the "has-a" relationship).
- When you establish "is-a" relationships between classes, you are building a dependency between types.
- The basic idea behind classical inheritance is that new classes may leverage (and possibly extend) the functionality of other classes.

C#'S INHERITANCE SUPPORT

- To illustrate, assume that you wish to leverage the functionality of the Employee class to create two new classes (SalesPerson and Manager).



C#'S INHERITANCE SUPPORT

- As illustrated in Figure 2-1, you can see that a SalesPerson "is-a" Employee (as is a Manager).
- In the classical inheritance model, base classes (such as Employee) are used to define general characteristics that are common to all descendents.
- Subclasses (such as SalesPerson and Manager) extend this general functionality while adding more specific behaviors.

C#'S INHERITANCE SUPPORT

- The general form of a **class** declaration that inherits a base class is shown here:

```
class derived-class-name : base-class-name {
// body of class
}
```

- **Program ch02pg11.cs:**
  - For example, the following program uses **TwoDShape** to derive a class called **Triangle**. **TwoDShape** can be used as a base class (that is, as a starting point) for classes that describe specific types of two-dimensional objects. Pay close attention to the way that **Triangle** is declared.

CONTROLLING BASE CLASS CREATION WITH BASE

- In a hierarchy, it is possible for both base classes and derived classes to have their own constructors.
- The constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part.
- A derived class can call a constructor defined in its base class by using an expanded form of the derived class' constructor declaration and the **base** keyword.

CONTROLLING BASE CLASS CREATION WITH BASE

- The general form of this expanded declaration is shown here:

```
derived-constructor(parameter-list) : base(arg-list)
{
// body of constructor
}
```

- Here, *arg-list* specifies any arguments needed by the constructor in the base class. Notice the placement of the colon.

CONTROLLING BASE CLASS CREATION WITH BASE

- In C#, the *default constructor* of a base class is called automatically before the derieved class constructor is executed.
- To call the parameterized constructor of a derived class, you need to explicitly call an appropriate custom base class constructor with the **base** keyword, rather than the default.
  - **Program ch01pg12.cs:** To see how **base** is used, consider the version of **TwoDShape** in the following program. It defines a constructor that initializes the **Width** and **Height** properties. This constructor is then called by the **Triangle** constructor.

REGARDING MULTIPLE BASE CLASSES

- Speaking of base classes, it is important to keep in mind that C# demands that a given class have *exactly one* direct base class.
- Therefore, it is not possible to have a single type with two or more base classes (this technique is known as *multiple inheritance*, or simply MI).
- C# does allow a given type to implement any number of discrete interfaces.
- In this way, a C# class can exhibit a number of behaviors while avoiding the problems associated with classic MI.

KEEPING FAMILY SECRETS: THE PROTECTED KEYWORD

- As you already know, public items are directly accessible from anywhere, while private items cannot be accessed from any object beyond the class that has defined it.
- C# takes the lead of many other modern object languages and provides an additional level of accessibility: protected .
- When a base class defines protected data or protected members; it is able to create a set of items that can be accessed directly by any descendent.

KEEPING FAMILY SECRETS: THE PROTECTED KEYWORD

- For example

```
// Protected state data.
public class Employee
{
// Child classes can directly access this information. Object users cannot.
protected string fullName;
protected int empID;
protected float currPay;
protected string empSSN;
protected int empAge;
...
}
```

- The benefit of defining protected members in a base class is that derived types no longer have to access the data using public methods or properties.

PREVENTING INHERITANCE: SEALED CLASSES

- As powerful and useful as inheritance is, sometimes you will want to prevent it.
- For example, you might have a class that encapsulates the initialization sequence of some specialized hardware device, such as a medical monitor.
- In this case, you don't want users of your class to be able to change the way the monitor is initialized, possibly setting the device incorrectly.
- Whatever the reason, in C# it is easy to prevent a class from being inherited by using the keyword **sealed**.
- To prevent a class from being inherited, precede its declaration with **sealed**.

PREVENTING INHERITANCE: SEALED CLASSES

- Here is an example of a **sealed** class:

```
sealed class A {
// ...
}
// The following class is illegal.
class B : A { // ERROR! Can't derive from class A
// ...
}
```

- As the comments imply, it is illegal for **B** to inherit **A** because **A** is declared as **sealed**.

PREVENTING INHERITANCE: SEALED CLASSES

- **Program ch02pg13.cs:** This program demonstrates the use of sealed keyword in inheritance. In this example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

- The result is an error message: 'MyDerivedC' cannot inherit from sealed class 'SealedClass'.

PROGRAMMING FOR CONTAINMENT/DELEGATION

- In the second pillar of OOP, let's examine the "has-a" relationship (also known as the *containment/delegation* model).
- Assume you have created a new class that models an employee benefits package:

```
// This type will function as a contained class.
public class BenefitPackage
{
// Assume we have other members that represent
// 401K plans, dental / health benefits and so on.
public double ComputePayDeduction()
{ return 125.0; }
}
```

- Obviously, it would be rather odd to establish an “is-a” relationship between the BenefitPackage class and the employee types.
- (Manager “is-a” BenefitPackage? I don’t think so).
- However, you would like to express the idea that each employee “has-a” BenefitPackage.
- To do so, you can update the Employee class definition as follows:

PROGRAMMING FOR CONTAINMENT/DELEGATION

```
// Employees now have benefits.
public class Employee
{
...
// Contain a BenefitPackage object.
protected BenefitPackage empBenefits = new
BenefitPackage();
}
```

- At this point, you have successfully contained another object.
- However, to expose the functionality of the contained object to the outside world requires *delegation*.

PROGRAMMING FOR CONTAINMENT/DELEGATION

- **Delegation** is simply the act of adding members to the containing class that make use of the contained object's functionality.
- For example, we could update the Employee class to expose the contained empBenefits object using a custom property as well as make use of its functionality internally using a new method named GetBenefitCost():

```
public class Employee
{
    protected BenefitPackage empBenefits = new
    BenefitPackage();
```

PROGRAMMING FOR CONTAINMENT/DELEGATION

```
    // Expose certain benefit behaviors of object.
    public double GetBenefitCost()
    {
    return empBenefits.ComputePayDeduction();
    }
    // Expose object through a custom property.
    public BenefitPackage Benefits
    {
    get { return empBenefits; }
    set { empBenefits = value; }
    }
}
```

PROGRAMMING FOR CONTAINMENT/DELEGATION

- In the following updated Main() method, notice how we can interact with the internal BenefitsPackage type defined by the Employee type:

```
static void Main(string[] args)
{
Manager mel;
mel = new Manager();
Console.WriteLine(mel.Benefits.ComputePayDeduction()
);
...
Console.ReadLine();
}
```

NESTED TYPE DEFINITIONS

- In C#, it is possible to define a type (enum, class, interface, struct, or delegate) directly within the scope of a class or structure.
- Such type is termed as *nested types*.
- When you have done so, the nested (or "inner") type is considered a *member* of the nesting (or "outer") class, and in the eyes of the runtime can be manipulated like any other member (fields, properties, methods, events, etc.)

NESTED TYPE DEFINITIONS

- The syntax used to nest a type is quite straightforward:

```
public class OuterClass
{
// A public nested type can be used by anybody.
public class PublicInnerClass {}
// A private nested type can only be used by members
// of the containing class.
private class PrivateInnerClass {}
}
```

QUICK SUMMARY

- Nesting types is similar to composition ("has-a"), except that you have complete control over the access level of the inner *type* instead of a contained *object*.
- Because a nested type is a member of the containing class, it can access private members of the containing class.
- Oftentimes, a nested type is only useful as helper for the outer class, and is not intended for use by the outside world.

C #'S POLYMORPHIC SUPPORT

- Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance.
- Polymorphism is a Greek word that means "many-shaped".
- One aspect of polymorphism is that Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation.

C #'S POLYMORPHIC SUPPORT

- At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes; which in turn override of the virtual method.
- Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

C #'S POLYMORPHIC SUPPORT

- **Virtual methods** enable you to work with groups of related objects in a uniform way.
- For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface.
- You do not know at compile time which specific types of shapes the user will create.
- However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions.

C #'S POLYMORPHIC SUPPORT

- You can use polymorphism to solve this problem in two basic steps:
  - Create a class hierarchy in which each specific shape class derives from a common base class.
  - Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.
- **Program ch02pg15.cs**

VIRTUAL METHODS

- When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class.
- The designer of the derived class can choose whether to
  - override virtual members in the base class,
  - inherit the closest base class method without overriding it
  - define new non-virtual implementation of those members that hide the base class implementations

VIRTUAL METHODS

- A derived class can override a base class member only if the base class member is declared as virtual or abstract.
- The derived member must use the override keyword to explicitly indicate that the method is intended to participate in virtual invocation.
- The following code provides an example:

VIRTUAL METHODS

```
public class BaseClass
{
    int x;
    public virtual void
DoWork() { }
    public virtual int
WorkProperty
    {
            set { x=value; }
       get { return x; }
    }
}
```

```
public class DerivedClass :
BaseClass
{
    public override void DoWork()
{ }
    public override int
WorkProperty
    {
        get { return 0; }
    }
}
```

VIRTUAL METHODS

- Fields cannot be virtual; only methods, properties, events and indexers can be virtual.
- When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class.
- The following code provides an example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

VIRTUAL METHODS

- **Program ch02pg16.cs:**
  - This program demonstrates the above discussed logic with a change.
  - In derived class we have not defined override method DoWork().
  - When we invoke the objects created as above, the first object B will call the derived class DoWork() with derived class WorkProperty.
  - The second object A calls the base class DoWork() with derived class WorkProperty.

REVISITING THE SEALED KEYWORD

- The sealed keyword can also be applied to type members to prevent virtual members from being further overridden by derived types.
- This can be helpful when you do not wish to seal an entire class, just a few select methods or properties.
- Sealing a method requires putting the sealed keyword before the override keyword in the class member declaration.
- The following code provides an example:

```
public class C : B{    public sealed override void
DoWork() { }}
```

UNDERSTANDING ABSTRACT CLASSES

- Classes can be declared as abstract by putting the keyword abstract before the class definition.
- For example:

```
public abstract class A
{
    // Class members here.
}
```

- An abstract class cannot be instantiated.
- The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

UNDERSTANDING ABSTRACT CLASSES

- For example, class A may be defined as abstract class, and then can be used to provide different implementation of it, by creating a derived class.
- Abstract classes may also define abstract methods.
- This is accomplished by adding the keyword abstract before the return type of the method.
- For example:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

UNDERSTANDING ABSTRACT CLASSES

- Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block.
- Derived classes of the abstract class must implement all abstract methods.
- When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.
- **Program ch02pg17.cs**

UNDERSTANDING ABSTRACT CLASSES

- If a virtual method is declared abstract, it is still virtual to any class inheriting from the abstract class.
- A class inheriting an abstract method cannot access the original implementation of the method—in the previous example; DoWork on class F cannot call DoWork on class D.
- In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

UNDERSTANDING C# PARTIAL TYPES

- C# introduces a new type modifer named partial that allows you to define a C# type across multiple *.cs files.
- Earlier versions of the C# programming language required all code for a given type be defined within a single *.cs file.
- Given the fact that a production-level C# class may be hundreds of lines of code (or more), this can end up being a mighty long file indeed.
- In these cases, it would be ideal to partition a type's implementation across numerous C# files in order to separate code that is in some way more important for other details.

UNDERSTANDING C# PARTIAL TYPES

- For example, using the partial class modifer, you could place all public members in a file named MyType_Public.cs, while the private field data and private helper functions are defined within MyType_Private.cs:

UNDERSTANDING C# PARTIAL TYPES

```
// MyClass_Private.cs
namespace PartialTypes
{
public partial class
MyClass
{
// Private field data.
private string
someStringData;
}
}
```

```
// MyClass_Public.cs
namespace PartialTypes
{
  public partial class MyClass
  {
  // Constructors.
  public MyClass() { }
  // All public members.
  public void MemberA() { }
  }
}
```

UNDERSTANDING C# PARTIAL TYPES

- As you might guess, this can be helpful to new team members who need to quickly learn about the public interface of the type.
- Rather than reading though a single (lengthy) C# file to find the members of interest, they can focus on the public members.
- Of course, once these files are compiled by csc.exe, the end result is a single unified type.
- **Program ch02pg18.cs**

DOCUMENTING C# SOURCE CODE VIA XML

- C# give us the ability to maintain code and documentation in the same file, which makes the whole process a lot easier.
- VS.NET does this by taking specially marked and structured comments from within the code and building them into an XML file.
- This XML file can then be used to generate human-readable documentation.

DOCUMENTING C# SOURCE CODE VIA XML

- When you wish to document your C# types in XML, your first step is to make use of one of two notations, the triple forward slash (///) or a delimited comment that begins with a single forward slash and two stars (/**) and ends with a single star-slash combo (*/).
- Once a documentation comment has been declared, you are free to use any well-formed XML elements, including the recommended set shown in Table 2-1

DOCUMENTING C# SOURCE CODE VIA XML

Predefined XML	Documentation Element Meaning in Life
<c>	Indicates that the following text should be displayed in a specific “code font”
<code>	Indicates multiple lines should be marked as code
<example>	Mocks up a code example for the item you are describing
<exception>	Documents which exceptions a given class may throw
<list>	Inserts a list or table into the documentation file
<param>	Describes a given parameter
<paramref>	Associates a given XML tag with a specific parameter

DOCUMENTING C# SOURCE CODE VIA XML

Predefined XML	Documentation Element Meaning in Life
<permission>	Documents the security constraints for a given member
<remarks>	Builds a description for a given member
<returns>	Documents the return value of the member
<see>	Cross- references related items in the document
<seealso>	Builds an “also see” section within a description
<summary>	Documents the “executive summary” for a given member
<value>	Documents a given property

DOCUMENTING C# SOURCE CODE VIA XML

- **Program ch02pg19.cs:** The following sample provides a basic overview of a type that has been documented.
- If you are building your C# programs using csc.exe, the /doc flag is used to generate a specified *.xml file based on your XML code comments:
- csc /doc:XmlCarDoc.xml *.cs

XML CODE COMMENT FORMAT CHARACTERS

- **XML Code Comment Format Characters**
  - If you were now to open the generated XML file, you will notice that the elements are qualified by numerous characters such as “M”, “T”, “F”, and so on.
  - For example:

```
<member name="T:XmlDocCar.Car">
<summary>
This is a simple Car that illustrates
working with XML style documentation.
</summary>
</member>
```

XML CODE COMMENT FORMAT CHARACTERS

Format	Character Meaning in Life
E	Item denotes an event.
F	Item represents a field.
M	Item represents a method (including constructors and overloaded operators).
N	Item denotes a namespace.
P	Item represents type properties (including indexes).
T	Item represents a type (e.g., class, interface, struct, enum, delegate).

UNDERSTANDING OBJECT LIFETIME CLASSES

- Classes, of course, are defined within a code file (which in C# takes a *.cs extension by convention).
- Consider a simple Car class defined within Car.cs:

```
// Car.cs
public class Car
{
.....
}
```

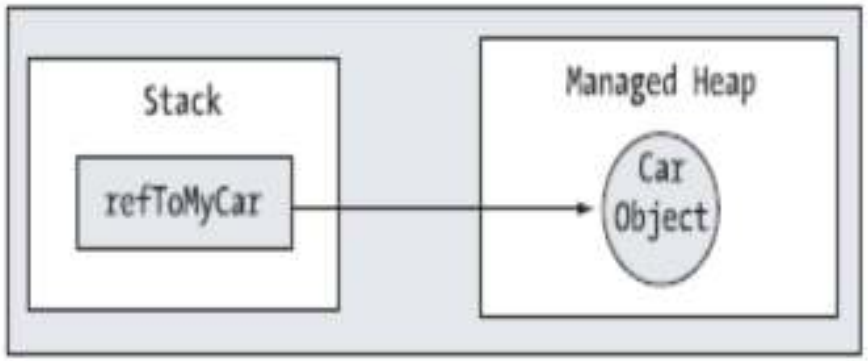
UNDERSTANDING OBJECT LIFETIME CLASSES

- Once a class is defined, you can allocate any number of objects using the C# new keyword.
- Understand, however, that the new keyword returns a *reference* to the object on the heap, not the actual object itself.
- This reference variable is stored on the stack for further use in your application.
- When you wish to invoke members on the object, apply the C# dot operator to the stored reference

UNDERSTANDING OBJECT LIFETIME CLASSES

- For example:

```
// The C# dot operator (.) is used
// to invoke members on the object
// using our reference variable.
Console.WriteLine(refToMyCar.ToString());
```



THE BASICS OF OBJECT LIFETIME

- When you are building your C# applications, you are correct to assume that the managed heap will take care of itself without your direct intervention.
- In fact, the golden rule of .NET memory management is simple:
- **Rule:** Allocate an object onto the managed heap using the new keyword and forget about it.
- Once created the garbage collector will destroy the object when it is no longer needed.

THE BASICS OF OBJECT LIFETIME

- The next obvious question, of course, is, “How does the garbage collector determine when an object is no longer needed”?
- The short (i.e., incomplete) answer is that the garbage collector removes an object from the heap when it is *unreachable* by any part of your code base.

THE CIL OF NEW

- When the C# compiler encounters the new keyword, it will emit a CIL newobj instruction into the method implementation.
- The .NET garbage collector is quite a tidy housekeeper of the heap, given that it will compact empty blocks of memory (when necessary) for purposes of optimization.
- Before we examine the exact rules that determine when an object is removed from the managed heap, let's check out the role of the CIL newobj instruction in a bit more detail.

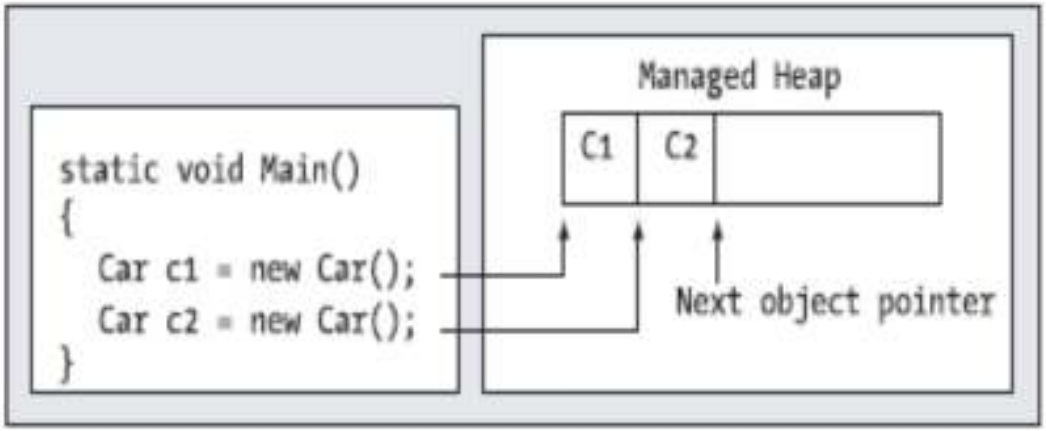
THE CIL OF NEW

- These things being said, the newobj instruction informs the CLR to perform the following core tasks:
  - Calculate the total amount of memory required for the object to be allocated (including the necessary memory required by the type's member variables and the type's base classes).
  - Examine the managed heap to ensure that there is indeed enough room to host the object to be allocated. If this is the case, the type's constructor is called, and the caller is ultimately returned a reference to the new object in memory, whose address just happens to be identical to the last position of the next object pointer.

THE CIL OF NEW

- Finally, before returning the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap.

- The basic process is illustrated in Figure below:



THE CIL OF NEW

- When processing the newobj instruction, if the CLR determines that the managed heap does not have sufficient memory to allocate the requested type, it will perform a garbage collection in an attempt to free up memory.
- Thus, the next rule of garbage collection is also quite simple.
- **Rule:** If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will occur.

THE CIL OF NEW

- When a collection does take place, the garbage collector temporarily suspends all active *threads* within the current process to ensure that the application does not access the heap during the collection process.

SYSTEM.GC TYPE

- The base class libraries provide a class type named System.GC that allows you to programmatically interact with the garbage collector using a set of static members.
- Typically speaking, the only time you will make use of the members of System.GC is when you are creating types that make use of *unmanaged resources*.
- Table 2 provides a rundown of some of the more interesting members:

SYSTEM.GC TYPE

System.GC Member	Meaning in Life
AddMemoryPressure(), RemoveMemoryPressure()	Allow you to specify a numerical value that represents the calling RemoveMemoryPressure() object's "urgency level" regarding the garbage collection process. Be aware that these methods should alter pressure in tandem and thus never remove more pressure than the total amount you have added.
Collect()	Forces the GC to perform a garbage collection

SYSTEM.GC TYPE

System.GC Member	Meaning in Life
GetGeneration()	Returns the generation to which an object currently belongs.
GetTotalMemory()	Returns the estimated amount of memory (in bytes) currently allocated on the managed heap.
MaxGeneration	Returns the maximum of generations supported on the target system. Under Microsoft's .NET, there are three possible generations (0, 1, and 2).
SuppressFinalize()	Sets a flag indicating that the specified object should not have its Finalize() method called.

BUILDING FINALIZABLE OBJECTS

- The supreme base class of .NET, System.Object, defines a virtual method named Finalize().
- When you override Finalize() for your custom classes, you establish a specific location to perform any necessary cleanup logic for your type.
- Given that this member is defined as protected, it is not possible to directly call an object's Finalize() method.
- Rather, the *garbage collector* will call an object's Finalize() method (if supported) before removing the object from memory.

BUILDING FINALIZABLE OBJECTS

- Of course, a call to Finalize() will (eventually) occur during a “natural” garbage collection or when you programmatically force a collection via GC.Collect().
- In addition, a type’s finalizer method will automatically be called when the *application domain* hosting your application is unloaded from memory.
- Given this, consider the next rule of garbage collection:
- **Rule:** The only reason to override Finalize() is if your C# class is making use of unmanaged resources via PInvoke or complex COM interoperability tasks .

BUILDING FINALIZABLE OBJECTS

- To override Finalize() is that you cannot do so using the expected override keyword:

```
public class MyResourceWrapper
{
// Compile time error!
protected override void Finalize(){ }
}
```

- Rather, you make use of the following (C++-like) destructor syntax to achieve the same effect.

BUILDING FINALIZABLE OBJECTS

```
// Override System.Object.Finalize() via destructor
syntax.
class MyResourceWrapper
{
  ~MyResourceWrapper()
  {
  // Clean up unmanaged resources here.
  // Beep when destroyed (testing purposes only!)
  Console.Beep();
  }
}
```

BUILDING DISPOSABLE OBJECTS

- As an alternative to overriding Finalize(), your class could implement the IDisposable interface, which defines a single method named Dispose():

```
public interface IDisposable
{
void Dispose();
}
```

- When you do support the IDisposable interface, the assumption is that when the *object user* is finished using the object, it manually calls Dispose() before allowing the object reference to drop out of scope.

BUILDING DISPOSABLE OBJECTS

- In this way, your objects can perform any necessary cleanup of unmanaged resources without incurring the hit of being placed on the finalization queue and without waiting for the garbage collector to trigger the class's finalization logic.
- Here is an updated MyResourceWrapper class that now implements IDisposable, rather than overriding System.Object.Finalize():

BUILDING DISPOSABLE OBJECTS

```
// Implementing IDisposable.
public class MyResourceWrapper : IDisposable
{
   // The object user should call this method
   // when they finished with the object.
   public void Dispose()
   {
   // Clean up unmanaged resources here.
   // Dispose other contained disposable objects.
   }
}
```

BUILDING DISPOSABLE OBJECTS

- The calling logic is straightforward:

```
public class Program
{
   static void Main()
   {
   MyResourceWrapper rw = new MyResourceWrapper();
   rw.Dispose();
   Console.ReadLine();
   }
}
```