msdn training

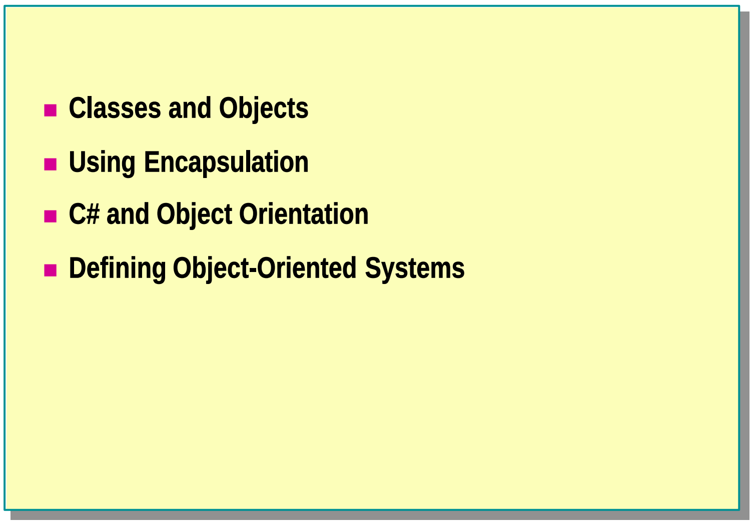
Module 7: Essentials of Object-Oriented Programming

Contents

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.

Microsoft

Overview



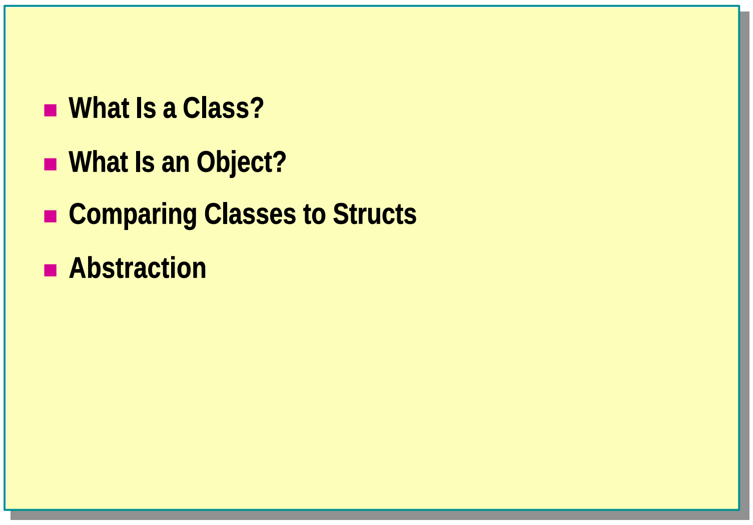
- Classes and Objects
- Using Encapsulation
- C# and Object Orientation
- Defining Object-Oriented Systems

C# is an object-oriented programming language. In this section, you will learn the terminology and concepts required to create and use classes in C#.

After completing this module, you will be able to:

- Define the terms *object* and *class* in the context of object-oriented programming.
- Define the three core aspects of an object: identity, state, and behavior.
- Describe abstraction and how it helps you to create reusable classes that are easy to maintain.
- Use encapsulation to combine methods and data in a single class.
- Explain the concepts of inheritance and polymorphism.
- Create and use classes in C#.

◆ Classes and Objects

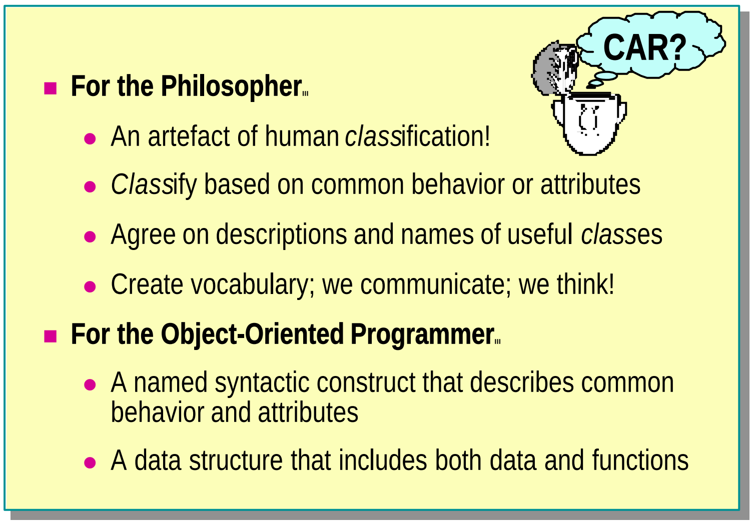


- What Is a Class?
- What Is an Object?
- Comparing Classes to Structs
- Abstraction

The whole structure of C# is based on the object-oriented programming model. To make the most effective use of C# as a language, you need to understand the nature of object-oriented programming.

In this section, you will learn about the basics of object-oriented programming. You will examine classes and objects in the context of object-oriented programming. You will then learn the how to apply the concept of abstraction.

What Is a Class?



- **For the Philosopher…**
  - An artefact of human *class*ification!
  - *Class*ify based on common behavior or attributes
  - Agree on descriptions and names of useful *class*es
  - Create vocabulary; we communicate; we think!
- **For the Object-Oriented Programmer…**
  - A named syntactic construct that describes common behavior and attributes
  - A data structure that includes both data and functions

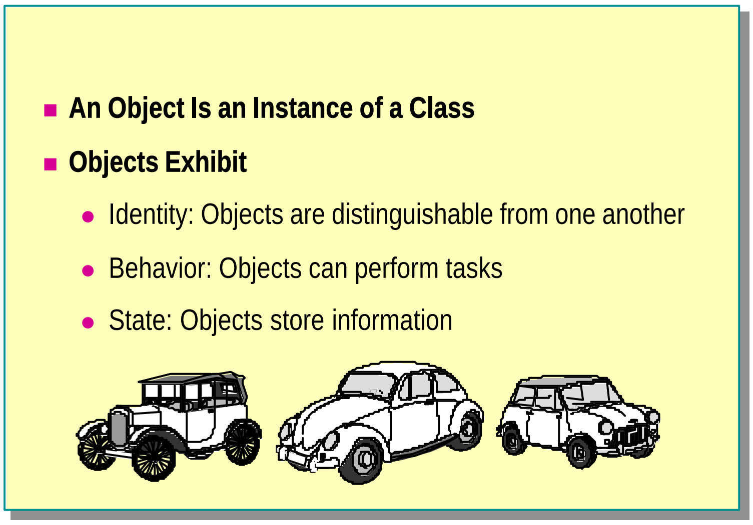
The root word of classification is *class*. Forming classes is an act of classification, and it is something that all human beings (not just programmers) do. For example, all cars share common behavior (they can be steered, stopped, and so on) and common attributes (they have four wheels, an engine, and so on). You use the word *car* to refer to all of these common behaviors and properties. Imagine what it would be like if you were not able to classify common behaviors and properties into named concepts! Instead of saying *car*, you would have to say all the things that *car* means. Sentences would be long and cumbersome. In fact, communication would probably not be possible at all. As long as everyone agrees what a word means, that is, as long as we all speak the same language, communication works well—we can express complex but precise ideas in a compact form. We then use these named concepts to form higher-level concepts and to increase the expressive power of communication.

All programming languages can describe common data and common functions. This ability to describe common features helps to avoid duplication. A key motto in programming is "Don't repeat yourself." Duplicate code is troublesome because it is more difficult to maintain. Code that does not repeat itself is easier to maintain, partly because there is just less of it! Object-oriented languages take this concept to the next level by allowing descriptions of classes (sets of objects) that share structure and behavior. If done properly, this paradigm works extremely well and fits naturally into the way people think and communicate.

Classes are not restricted to classifying concrete objects (such as cars); they can also be used to classify abstract concepts (such as time). However, when you are classifying abstract concepts, the boundaries are less clear, and good design becomes more important.

The only real requirement for a class is that it helps people communicate.

What Is an Object?



- **An Object Is an Instance of a Class**
- **Objects Exhibit**
  - Identity: Objects are distinguishable from one another
  - Behavior: Objects can perform tasks
  - State: Objects store information

The word *car* means different things in different contexts. Sometimes we use the word car to refer to the general concept of a car: we speak of car as a *class*, meaning the set of all cars, and do not have a specific car in mind. At other times we use the word car to mean a specific car. Programmers use the term *object* or *instance* to refer to a specific car. It is important to understand this difference.

The three characteristics of identity, behavior, and state form a useful way to think about and understand objects.

Identity

Identity is the characteristic that distinguishes one object from all other objects of the same class. For example, imagine that two neighbors own a car of exactly the same make, model, and color. Despite the obvious similarities, the registration numbers are guaranteed to be unique and are an outward reflection that cars exhibit identity. The law determines that it is necessary to distinguish one car object from another. (How would car insurance work without car identity?)

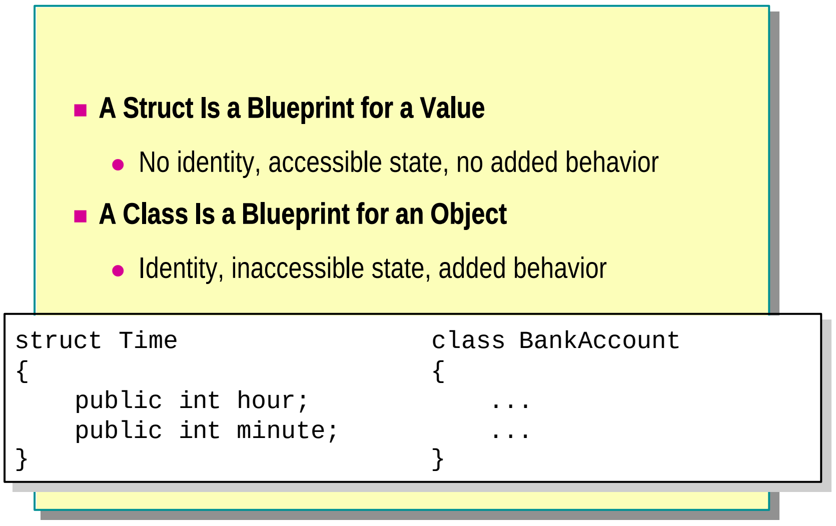
Behavior

Behavior is the characteristic that makes objects useful. Objects exist in order to provide behavior. Most of the time you ignore the workings of the car and think about its high-level behavior. Cars are useful because you can drive them. The workings exist but are mostly inaccessible. It is the behavior of an object that is accessible. The behavior of an object also most powerfully determines its classification. Objects of the same class share the same behavior. A car is a car because you can drive it; a pen is a pen because you can write with it.

State

State refers to the inner workings of an object that enable it to provide its defining behavior. A well-designed object keeps its state inaccessible. This is closely linked to the concepts of abstraction and encapsulation. You do not care how an object does what it does; you just care that it does it. Two objects may coincidentally contain the same state but nevertheless be two different objects. For example, two identical twins contain exactly the same state (their DNA) but are two distinct people.

Comparing Classes to Structs



- **A Struct Is a Blueprint for a Value**
  - No identity, accessible state, no added behavior
- **A Class Is a Blueprint for an Object**
  - Identity, inaccessible state, added behavior

```
struct Time                         class BankAccount
{                                   {
    public int hour;                    ...
    public int minute;                  ...
}                                   }
```

Structs

A struct, such as *Time* in the preceding code, has no identity. If you have two *Time* variables both representing the time 12:30, the program will behave exactly the same regardless of which one you use. Software entities with no identity are called *values*. The built-in types described in Module 3, " Using Value-Type Variables," in Course 2124A: *Introduction to C# Programming for the Microsoft .NET Platform (Prerelease),* such as int, bool, decimal, and all struct types, are called *value types* in C#. Value types contain accessible state and have no added behavior (no methods).

Variables of the struct type are allowed to contain methods, but it is recommended that they do not. They should contain only data. However, it is perfectly reasonable to define operators in structs. Operators are stylized methods that do not add new behavior; they only provide a more concise syntax for existing behavior.

Classes

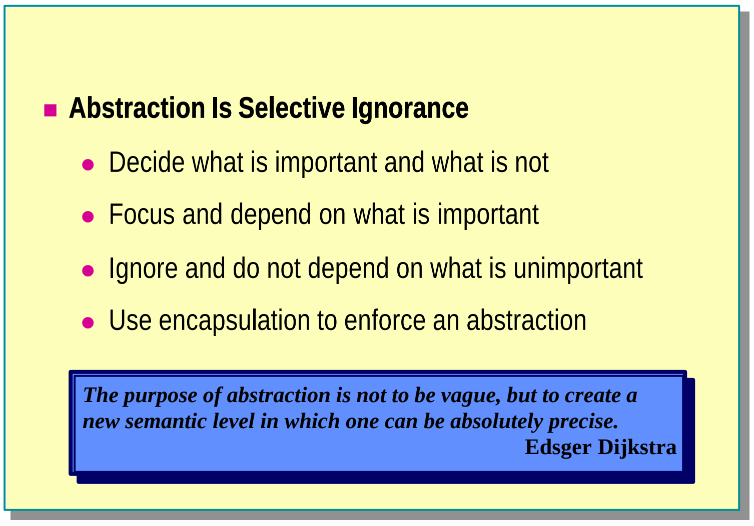
A class, such as **BankAccount** in the preceding code, has identity. If you have two **BankAccount** objects, the program will behave differently depending on which one you use. Software entities that have identity are called *objects*. (Variables of the struct type are also sometimes loosely called objects, but strictly speaking they are *values*.) Types represented by classes are called *reference types* in C#. In contrast to structs, nothing but methods should be visible in a well-designed class. These methods add extra high-level behavior beyond the primitive behavior present in the lower-level inaccessible data.

Value Types and Reference Types

Value types are the types found at the lowest level of a program. They are the elements used to build larger softw are entities. Value types can be freely copied and exist on the stack as local variables or as attributes inside the objects they describe.

Reference types are the types found at the higher levels of a program. They are built from smaller software entities. Reference types generally cannot be copied, and they exist on the heap.

Abstraction



- **Abstraction Is Selective Ignorance**
  - Decide what is important and what is not
  - Focus and depend on what is important
  - Ignore and do not depend on what is unimportant
  - Use encapsulation to enforce an abstraction

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
Edsger Dijkstra

Abstraction is the tactic of stripping an idea or object of its unnecessary accompaniments until you are left with its essential, minimal form. A good abstraction clears away unimportant details and allows you to focus and concentrate on the important details.

Abstraction is an important software principle. A well-designed class exposes a minimal set of carefully considered methods that provide the essential behavior of the class in an easy-to-use manner. Unfortunately, creating good software abstractions is not easy. Finding good abstractions usually requires a deep understanding of the problem and its context, great clarity of thought, and plenty of experience.

Minimal Dependency

The best software abstractions make complex things simple. They do this by ruthlessly hiding away unessential aspects of a class. These unessential aspects, once truly hidden away, cannot then be seen, used, or depended upon in any way.

It is this principle of minimal dependency that makes abstraction so important. One of the few things guaranteed in software development is that the code will need to be changed. Perfect understanding only comes at the end of the development process, if it comes at all; early decisions will be made with an incomplete understanding of the problem and will need to be revisited. Specifications will also change when a clearer understanding of the problem is reached. Future versions will require extra functionality. Change is normal in software development. The best you can do is to minimize the impact of change when it happens. And the less you depend on something, the less you are affected when it changes.

Related Quotes

To illustrate the principle of minimal dependency that makes abstraction so important, here are some related quotes:

The more perfect a machine becomes, the more they are invisible behind their function. It seems that perfection is achieved not when there is nothing more to add, but when there is nothing more to take away. At the climax of its evolution, the machine conceals itself entirely.

—Antoine de Saint-Exupéry, *Wind, Sand and Stars*

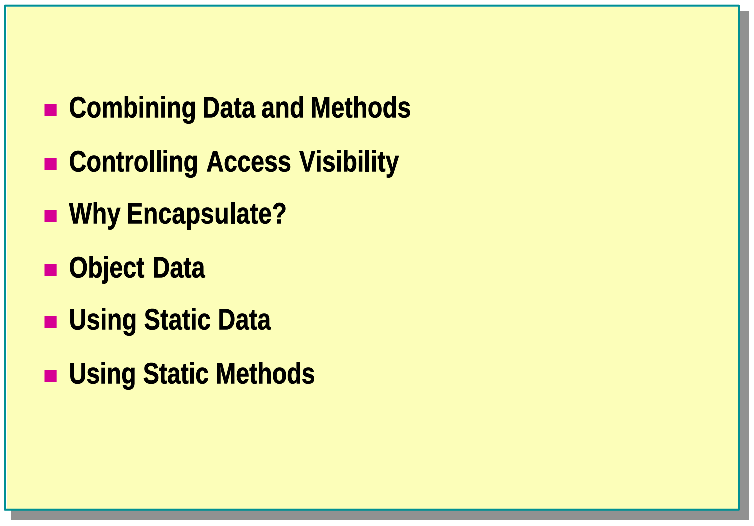
The minimum could be defined as the perfection that an artifact achieves when it is no longer possible to improve it by subtraction. This is the quality that an object has when every component, every detail, and every junction has been reduced or condensed to the essentials. It is the result of the omission of the inessentials.

—John Pawson, *Minimum*

The main aim of communication is clarity and simplicity. Simplicity means focused effort.

—Edward de Bono, *Simplicity*

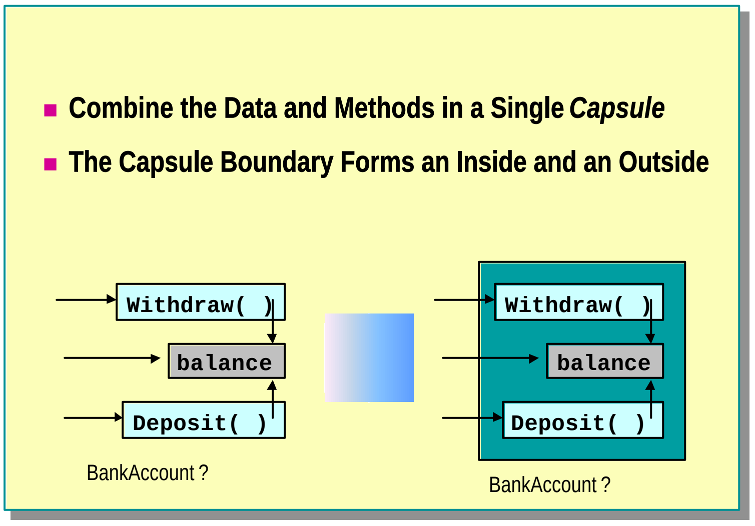
◆ Using Encapsulation



- Combining Data and Methods
- Controlling Access Visibility
- Why Encapsulate?
- Object Data
- Using Static Data
- Using Static Methods

In this section, you will learn how to combine data and methods in a single capsule. You will learn how to use encapsulation within a class, and you will also learn how to use static data methods in a class.

Combining Data and Methods



- Combine the Data and Methods in a Single *Capsule*
- The Capsule Boundary Forms an Inside and an Outside

There are two important aspects to encapsulation:

- Combining data and functions in a single entity (covered in the slide)
- Controlling the accessibility of the entity members (covered in the next slide)

Procedural Programming

Traditional procedural programs written in languages such as C essentially contain a lot of data and many functions. Every function can access every piece of data. For a small program this highly coupled approach can work, but as the program grows larger it becomes less feasible. Changing the data representation causes havoc. All functions that use (and hence depend upon) the changed data fail. As the program becomes larger, making any change becomes more difficult. The program becomes more brittle and less stable. The separate data-function approach does not scale. It does not facilitate change, and as all software developers know, change is the only constant.

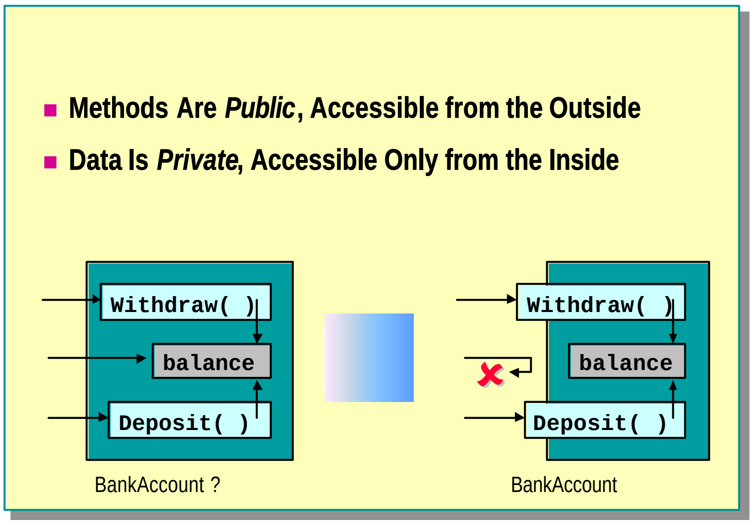
There is another serious problem with keeping the data separated from the functions. This technique does not correspond to the way people naturally think, in terms of high-level behavioral abstractions. Because people (the ones who are programmers) write programs, it is much better to use a programming model that approximates the way people think rather than the way computers are currently built.

Object-Oriented Programming

Object-oriented programming arose to alleviate these problems. Object-oriented programming, if understood and used wisely, is really person-oriented programming because people naturally think and work in terms of the high-level behavior of objects.

The first and most important step away from procedural programming and towards object-oriented programming is to combine the data and the functions into a single entity.

Controlling Access Visibility



- Methods Are *Public*, Accessible from the Outside
- Data Is *Private*, Accessible Only from the Inside

In the graphic on the left, **Withdraw**, **Deposit**, and **balance** have been grouped together inside a “capsule.” The slide suggests that the name of the capsule is **BankAccount**. However, there is something wrong with this model of a bank account: the **balance** data is accessible. (Imagine if real bank account balances were directly accessible like this; you could increase your balance without making any deposits!) This is not how bank accounts work: the problem and its model have poor correspondence.

You can solve this problem by using encapsulation. Once data and functions are combined into a single entity, the entity itself forms a closed boundary, naturally creating an inside and an outside. You can use this boundary to selectively control the accessibility of the entities: some will be accessible only from the inside; others will be accessible from both the inside and the outside. Those members that are always accessible are *public*, and those that are only accessible from the inside are *private*. It is not possible to have members that are only accessible from the outside.

To make the model of a bank account closer to a real bank account, you can make the **Withdraw** and **Deposit** methods public, and the **balance** private. Now the only way to increase the account balance from the outside is to deposit some money into the account. Note that **Deposit** can access the **balance** because **Deposit** is on the inside.

C#, like many other object-oriented programming languages, gives you complete freedom when choosing whether to make members accessible. You can, if you want, create public data. However, it is recommended that data always be marked private. (Some programming languages enforce this guideline.)

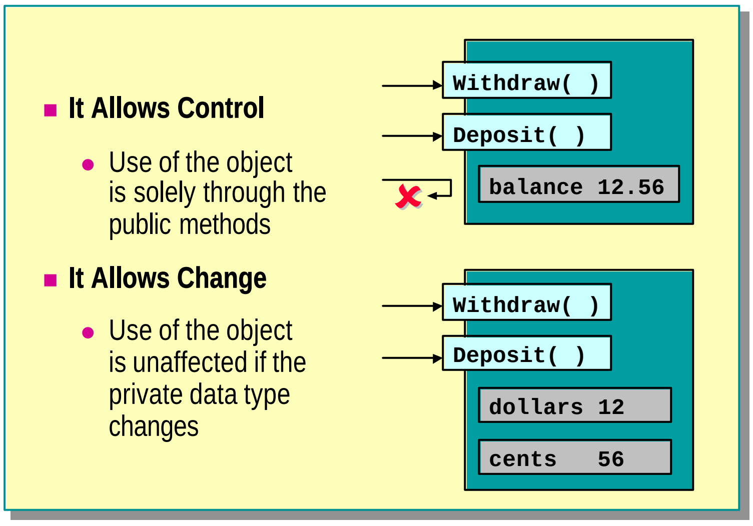
Types whose data representation is completely private are called abstract data types (ADTs). They are abstract in the sense that you cannot access (and rely on) the private data representation; you can only use the behavioral methods.

The built-in types such as **int** are, in their own way, ADTs. When you want to add two integer variables together, you do not need to know the internal binary representation of each integer value; you only need to know the name of the method that performs addition: the addition operator (+).

When you make members accessible (public), you can create different views of the same entity. The view from the outside is a subset of the view from the inside. A restricted view relates closely to the idea of abstraction: stripping an idea down to its essence.

A lot of design is related to the decision of whether to place a feature on the inside or on the outside. The more features you can place on the inside (and still retain usability) the better.

Why Encapsulate?


- **It Allows Control**
  - Use of the object is solely through the public methods
- **It Allows Change**
  - Use of the object is unaffected if the private data type changes

Two reasons to encapsulate are:

- To control use.
- To minimize the impact of change.

Encapsulation Allows Control

The first reason to encapsulate is to control use. When you drive a car, you think only about the act of driving, not about the internals of the car. When you withdraw money from an account, you do not think about how the account is represented. You can use encapsulation and behavioral methods to design software objects so that they can only be used in the way you intend.

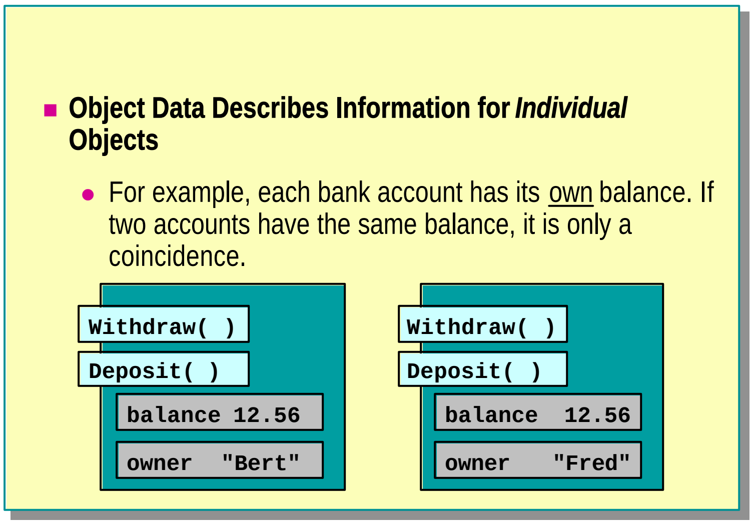
Encapsulation Allows Change

The second reason to encapsulate follows from the first. If an object's implementation detail is private, it can be changed and the changes will not directly affect users of the object (who can only access the public methods). In practice, this can be tremendously useful. The names of the methods typically stabilize well before the implementation of the methods.

The ability to make internal changes links closely to abstraction. Given two designs for a class, as a rule of thumb, use the one with fewer public methods.

In other words, if you have a choice about whether to make a method public or private, make it private. A private method can be freely changed and perhaps later promoted into a public method. But a public method cannot be demoted into a private method without destroying client code.

Object Data



- **Object Data Describes Information for *Individual* Objects**
  - For example, each bank account has its <u>own</u> balance. If two accounts have the same balance, it is only a coincidence.

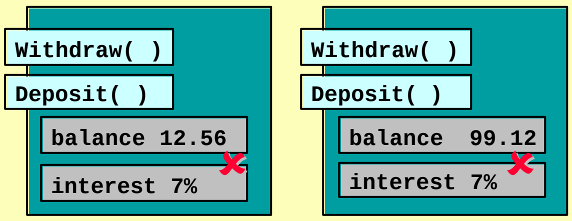
Withdraw( )
Deposit( )
balance 12.56
owner "Bert"
Withdraw( )
Deposit( )
balance 12.56
owner "Fred"

Most items of data inside an object describe information about that individual object. For example, each bank account has its own balance. It is, of course, perfectly possible for many bank accounts to have the same balance. However, this would only be a coincidence.

The data inside an object is held privately, and is accessible only to the object methods. This encapsulation and separation means that an object is effectively self-contained.

Using Static Data

- **Static Data Describes Information for *All* Objects of a Class**
  - For example, suppose all accounts share the same interest rate. Storing the interest rate in every account would be a bad idea. Why?



Sometimes it does not make sense to store information inside every object. For example, if all bank accounts always share the same interest rate, then storing the rate inside every account object would be a bad idea for the following reasons:

- It is a poor implementation of the problem as described: “All bank accounts share the same interest rate.”
- It needlessly increases the size of each object, using extra memory resources when the program is running and extra disk spac e when it is saved to disk.
- It makes it difficult to change the interest rate. You would need to change the interest rate in every account object. If you needed to make the interest rate change in each individual object, an interest rate change might make all accounts inaccessible while the change took place.
- It increases the size of the class. The private interest rate data would require public methods. The account class is starting to lose its cohesiveness. It is no longer doing one thing and one thing well.

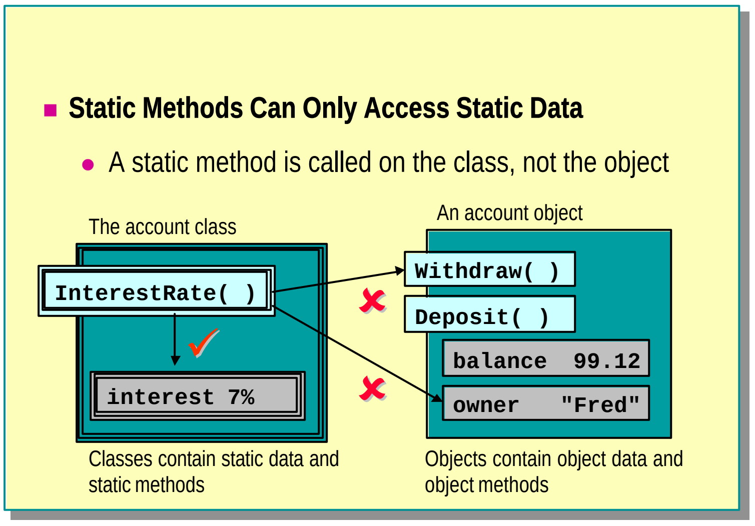
To solve this problem, do not share information that is common between objects at the object level. Instead of describing the interest rate many times at the object level, describe the interest rate once at the class level. When you define the interest rate at the class level, it effectively becomes global data.

However, global data, by definition, is not stored inside a class, and therefore cannot be encapsulated. Because of this, many object-oriented programming languages (including C#) do not allow global data. Instead, they allow data to be described as static.

Declaring Static Data

Static data is physically declared inside a class (which is a static, compile-time entity) and benefits from the encapsulation the class affords, but it is logically associated with the class itself and not with each object. In other words, static data is declared inside a class as a syntactic convenience and exists even if the program never creates any objects of that class.

Using Static Methods



- **Static Methods Can Only Access Static Data**
  - A static method is called on the class, not the object

You use static methods to encapsulate static data. In the example in the slide, the interest rate belongs to the account class and not to an individual account object. It therefore makes sense to provide methods at the class level that can be used to access or modify the interest rate.

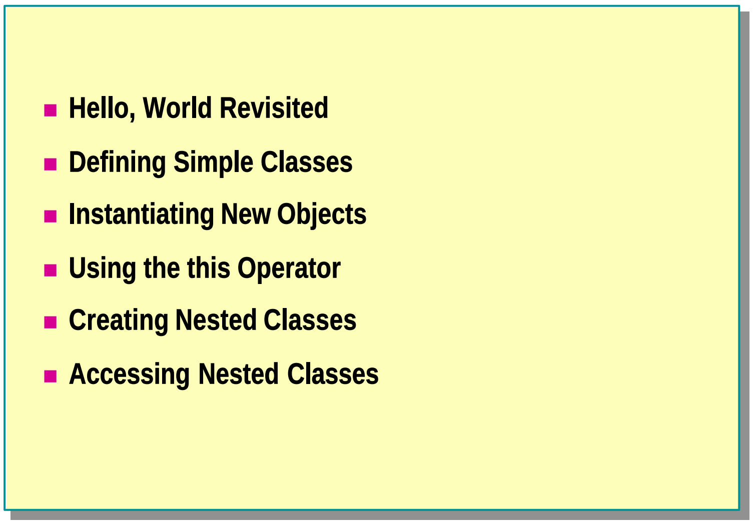
You can declare methods as static in the same way that you would declare data as static. Static methods exist at the class level. You can control accessibility for both static methods and static data can by using access modifiers such as public and private. By choosing public static methods and private static data, you can encapsulate static data in the same way that you can encapsulate object data.

A static method exists at the class level and is called against the class and not against an object. This means that a static method cannot use **this**, the operator that implicitly refers to the object making an object method call. In other words, a static method cannot access non-static data or non-static methods. The only members of a class that a static method can access are static data and other static methods.

Static methods retain access to all private members of a class and can access private non-static data by means of an object reference. The following code provides an example:

```
class Time
{
    ...
    public static void Reset(Time t)
    {
        t.hours = 0;     // Okay
        t.minutes = 0;   // Okay
        hour = 0;        // compile-time error
        minute = 0       // compile-time  error
    }
    private int hour, minute;
}
```

◆ C# and Object Orientation

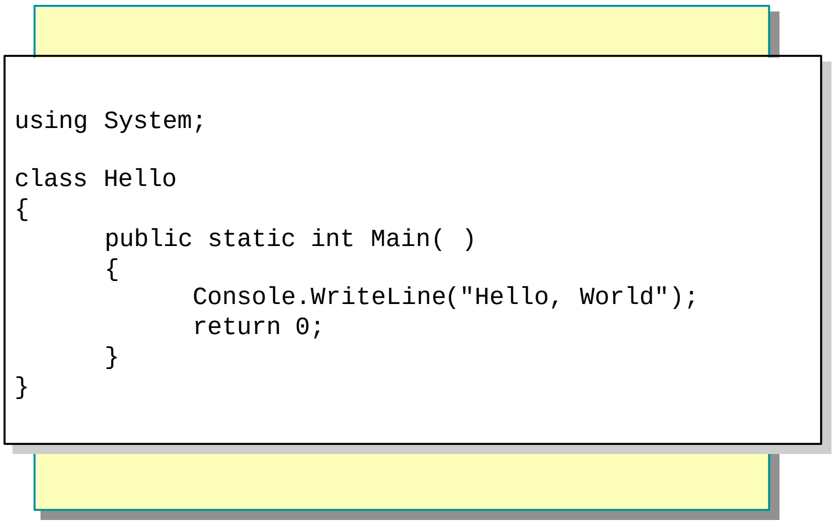


- Hello, World Revisited
- Defining Simple Classes
- Instantiating New Objects
- Using the this Operator
- Creating Nested Classes
- Accessing Nested Classes

In this section, you will re-examine the original Hello, World program.

The structure of the program will be explained from an object-oriented perspective. You will then learn about the mechanisms that enable one object to create another in C#. You will also learn how to define nested classes.

Hello, World Revisited



```
using System;

class Hello
{
      public static int Main( )
      {
            Console.WriteLine("Hello, World");
            return 0;
      }
}
```

The code for Hello, World is shown in the slide. There are some questions that can be asked and answered:

- How does the runtime invoke a class?
- Why is **Main** static?

How Does the Runtime Invoke a Class?

If there is a single **Main** method, the compiler will automatically make it the program entry point. The following code provides an example:

```
// OneEntrance.cs
class OneEntrance
{
    static void Main( )
    {
        ...
    }
}
// end of file

c:\> csc OneEntrance.cs
```

Warning The entry point of a C# program must be **Main** with a capital “M.” The signature of **Main** is also important.

However, if there are several methods called **Main**, one of them must explicitly be designated as the program entry point (and that **Main** must also be explicitly public) The following code provides an example:

```
// TwoEntries.cs
using System;
class EntranceOne
{
    public static void Main( )
    {
        Console.Write("EntranceOne.Main( )");
    }
}
class EntranceTwo
{
    public static void Main( )
    {
        Console.Write("EntranceTwo.Main( )");
    }
}
// End of file

c:\> csc /main:EntranceOne TwoEntries.cs
c:\> twoentries.exe
EntranceOne.Main( )
c:\> csc /main:EntranceTwo TwoEntries.cs
c:\> twoentries.exe
EntranceTwo.Main( )
c:\>
```

Note that the command-line option is case sensitive. If the name of the class containing **Main** is **EntranceOne** (with a capital E and a capital O) then the following will not work:

```
c:\> csc /main:entranceone TwoEntries.cs
```

If there is no **Main** method in the project, you cannot create an executable program. However, you can create a dynamic-link library (DLL) as follows:

```
// NoEntrance.cs
class NoEntrance
{
    public static void NotMain( )
    {
        Console.Write("NoEntrance.NotMain( )");
    }
}
// End of file

c:\> csc /target:library NoEntrance.cs
c:\> dir
...
NoEntrance.dll
...
```

Why Is Main Static?

Making **Main** static allows it to be invoked without the runtime needing to create an instance of the class.

Non-static methods can only be called on an object, as shown in the following code:

```
class Example
{
    void NonStatic( ) { ... }
    static void Main( )
    {
        Example eg = new Example( );
        eg.NonStatic( );  // Compiles
        NonStatic( );     // compile-time error
    }
    ...
}
```

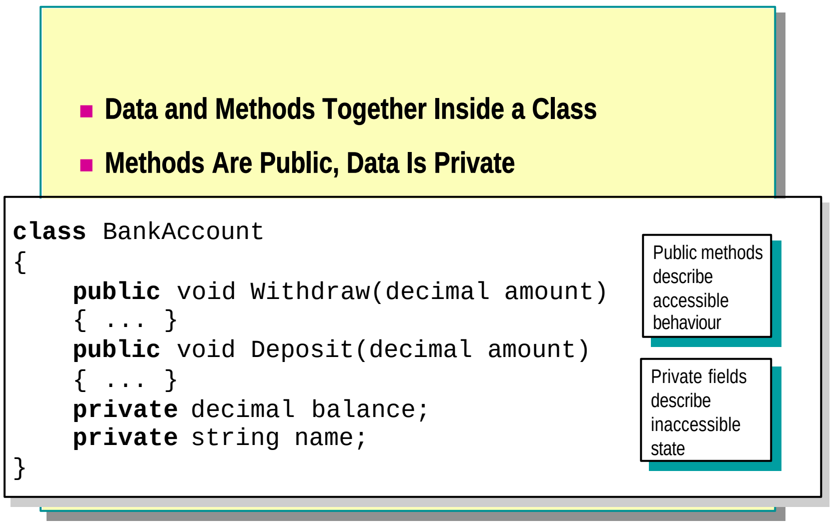
This means that if **Main** is non-static, as in the following code, the runtime needs to create an object in order to call **Main**.

```
class Example
{
    void Main( )
    {
        ...
    }
}
```

In other words, the runtime would effectively need to execute the following code:

```
Example run = new Example( );
run.Main( );
```

Defining Simple Classes



- Data and Methods Together Inside a Class
- Methods Are Public, Data Is Private

```
class BankAccount
{
    public void Withdraw(decimal amount)
    { ... }
    public void Deposit(decimal amount)
    { ... }
    private decimal balance;
    private string name;
}
```

Public methods describe accessible behaviour

Private fields describe inaccessible state

Although classes and structs are semantically different, they do have syntactic similarity. To define a class rather than a struct:

- Use the keyword **class** instead of **struct**.
- Declare your data inside the class exactly as you would for a struct.
- Declare your methods inside the class.
- Add access modifiers to the declarations of your data and methods. The simplest two access modifiers are **public** and **private**. (The other three will be covered later in this course.)

Note It is up to you to use **public** and **private** wisely to enforce encapsulation. C# does not prevent you from creating public data.

The meaning of *public* is "access not limited." The meaning of *private* is "access limited to the containing type." The following example clarifies this:

```
class BankAccount
{
    public void Deposit(decimal amount)
    {
        balance += amount;
    }
    private decimal balance;
}
```

In this example, the **Deposit** method can access the private **balance** because **Deposit** is a method of **BankAccount** (the type that contains **balance**). In other words, **Deposit** is on the inside. From the outside, private members are always inaccessible. In the following example, the expression `underAttack.balance` will fail to compile.

```
class BankRobber
{
    public void StealFrom(BankAccount underAttack)
    {
        underAttack.balance -= 999999M;
    }
}
```

The expression `underAttack.balance` will fail to compile because the expression is inside the **StealFrom** method of the **BankRobber** class. Only methods of the **BankAccount** class can access private members of **BankAccount** objects.

To declare static data, follow the pattern used by static methods (such as **Main**), and prefix the data declaration with the keyword **static**. The following code provides an example:

```
class BankAccount
{
    public void Deposit(decimal amount) { ... }
    public static void Main( ) { ... }
    ...
    private decimal balance;
    private static decimal interestRate;
}
```

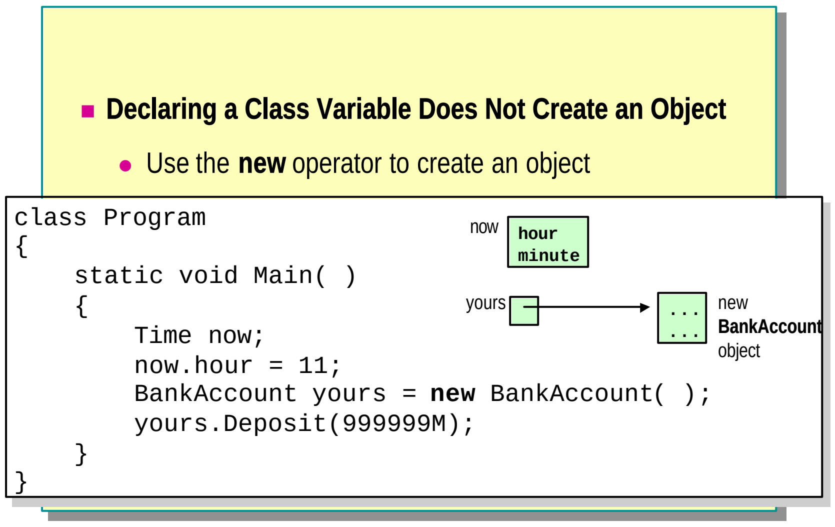
If you do not specify an access modifier when declaring a class member, it will default to private. In other words, the following two methods are semantically identical:

```
class BankAccount
{
    ...
    decimal balance;
}

class BankAccount
{
    ...
    private decimal balance;
}
```

Tips It is considered good style to explicitly write **private** even though it is not strictly necessary.

The order in which members of a class are declared is not significant to the C# compiler. However, it is considered good style to declare the public members (methods) before the private members (data). This is because a class user only has access to the public members anyway, and declaring public members before private members naturally reflects this priority.

Instantiating New Objects



- **Declaring a Class Variable Does Not Create an Object**
  - Use the **new** operator to create an object

```
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        BankAccount yours = new BankAccount( );
        yours.Deposit(999999M);
    }
}
```

now
hour
minute
yours
...
...
new
BankAccount
object

Consider the following code examples:

```
struct Time
{
    public int hour, minute;
}
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

Variables of the struct type are *value types*. This means that when you declare a struct variable (such as *now* in **Main**), you create a value on the stack. In this case, the *Time* struct contains two ints, so the declaration of *now* creates two **int**s on the stack, one called *now.hour* and one called *now.minute*. These two ints are not, repeat not, default initialized to zero. Hence the value of *now.hour* or *now.minute* cannot be read until they have been assigned a definite value. Values are scoped to the block in which they are declared. In this example, *now* is scoped to **Main**. This means that when the control flow exits **Main** (either through a normal return or because an exception has been thrown), *now* will go out of scope; it will cease to exist.

Classes are completely different as shown in the following code:

```
class Time // NOTE: Time is now a class
{
    public int hour, minute;
}
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

When you declare a class variable, you do not create an instance or object of that class. In this case, the declaration of *now* does not create an object of the **Time** class. Declaring a class variable creates a reference that is capable of referring to an object of that class. This is why classes are called *reference types*. This means that if the runtime were allowed to run the preceding code, it would be trying to access the integers inside a non-existent *Time* object. Fortunately, the compiler will warn you about this error. If you compile the preceding code, you will get the following error message:

```
error CS0165: Use of possibly unassigned local variable 'now'
```

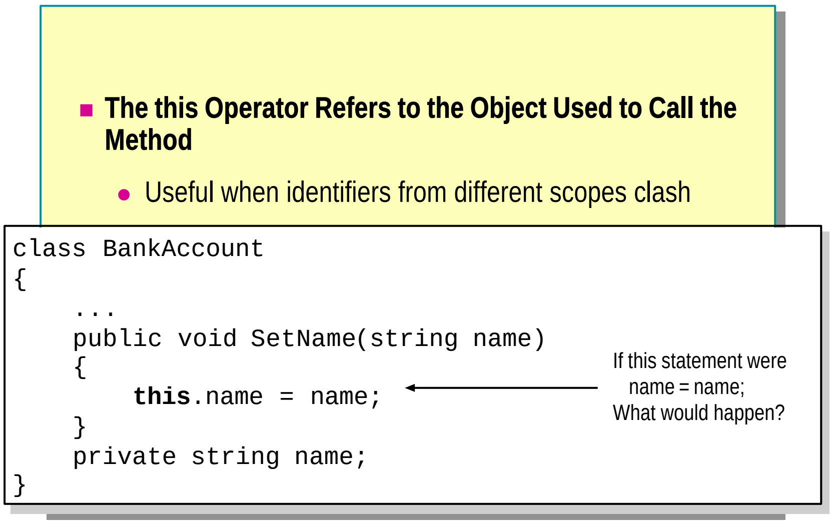
To fix this error, you must create a *Time* object (using the **new** keyword) and make the reference variable *now* actually refer to the newly created object, as in the following code:

```
class Program
{
    static void Main( )
    {
        Time now = new Time( );
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

Recall that when you create a local struct value on the stack, the fields are not, repeat not, default initialized to zero. Classes are different: when you create an object as an instance of a class, as above, the fields of the object are default initialized to zero. Hence the following code compiles cleanly:

```
class Program
{
    static void Main( )
    {
        Time now = new Time( );
        Console.WriteLine(now.hour);   // writes 0
        Console.WriteLine(now.minute); // writes 0
        ...
    }
}
```

Using the this Operator



- **The this Operator Refers to the Object Used to Call the Method**
  - Useful when identifiers from different scopes clash

```
class BankAccount
{
    ...
    public void SetName(string name)
    {
        this.name = name;
    }
    private string name;
}
```

If this statement were
name = name;
What would happen?

The **this** operator implicitly refers to the object that is making an object method call.

In the following code, the statement `name = name` would have no effect at all. This is because the identifier *name* on the left side of the assignment does not resolve to the private **BankAccount** field called *name*. Both identifiers resolve to the method parameter, which is also called *name*.

```
class BankAccount
{
    public void SetName(string name)
    {
        name = name;
    }
    private string name;
}
```

Warning The C# compiler does *not* emit a warning for this bug.

Using the this Keyword

You can solve this reference problem by using the **this** keyword, as illustrated on the slide. The **this** keyword refers to the current object for which the method is called.

Note Static methods cannot use **this** as they are not called by using an object.

Changing the Parameter Name

You can also solve the reference problem by changing the name of the parameter, as in the following example:

```
class BankAccount
{
    public void SetName(string newName)
    {
        name = newName;
    }
    private string name;
}
```

Tip Using **this** when writing constructors is a common C# idiom. The following code provides an example:

```
struct Time
{
    public Time(int hour, int minute)
    {
        this.hour = hour;
        this.minute = minute;
    }
    private int hour, minute;
}
```

Tip The **this** operator is also used to implement call chaining. Notice in the following class that both methods return the calling object:

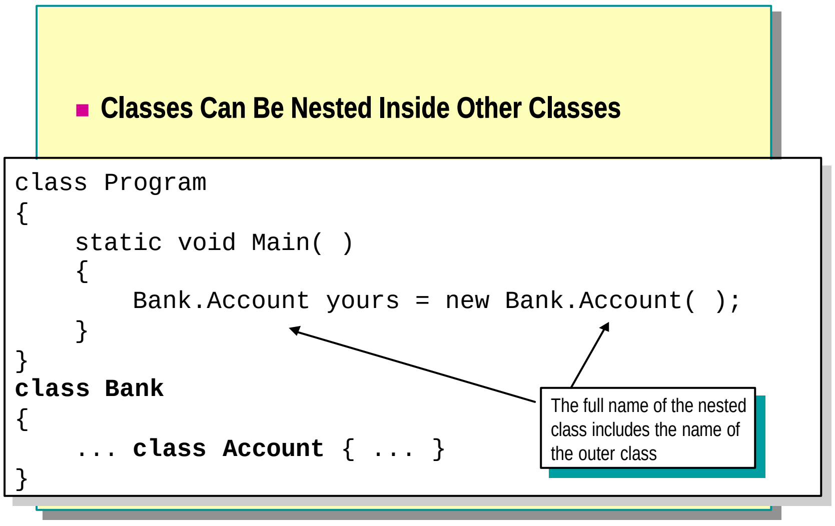
```
class Book
{
    public Book SetAuthor(string author)
    {
        this.author = author;
        return this;
    }
    public Book SetTitle(string title)
    {
        this.title = title;
        return this;
    }
    private string author, title;
}
```

Returning **this** allows method calls to be chained together, as follows:

```
class Usage
{
    static void Chained(Book good)
    {
        good.SetAuthor("Fowler").SetTitle("Refactoring");
    }
    static void NotChained(Book good)
    {
        good.SetAuthor("Fowler");
        good.SetTitle("Refactoring");
    }
}
```

Note A static method exists at the class level and is called against the class and not against an object. This means that a static method cannot use the **this** operator.

Creating Nested Classes



- Classes Can Be Nested Inside Other Classes

```
class Program
{
    static void Main( )
    {
        Bank.Account yours = new Bank.Account( );
    }
}
class Bank
{
    ... class Account { ... }
}
```

The full name of the nested class includes the name of the outer class

There are five different kinds of types in C#:

- class
- struct
- interface
- enum
- delegate

You can nest all five of these inside a class or a struct.

Note You cannot nest a type inside an interface, an enum, or a delegate.

In the code above, the **Account** class is nested inside the **Bank** class. The full name of the nested class is **Account . Bank**, and this name must be used when naming the nested type outside the scope of **Bank**. The following code provides an example:

```
// Program.cs
class Program
{
    static void Main( )
    {
        Account yours = new Account( ); // compile-time error
    }
}
// end of file
c:\> csc Program.cs
error CS0234: The type...'Account' does not exist in the
class...'Program'
```

In contrast, just the name **Account** can be used from inside of **Bank**, as in the following example:

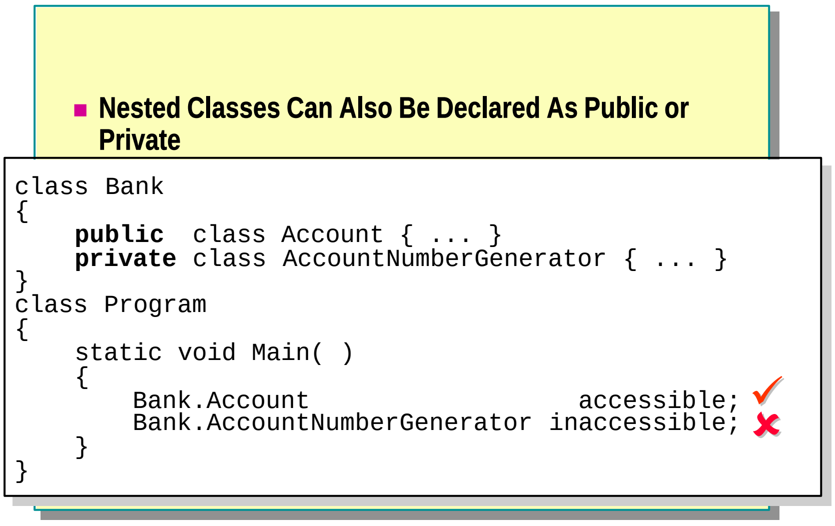
```
class Bank
{
    class Account( ) { ... }

    Account OpenAccount( )
    {
        return new Account( );
    }
}
```

Note See the next topic for a more thorough examination of the example.

Nested classes offer several useful features:

- Nested classes can be declared with specific accessibility. This is covered in the next topic.
- Using nested classes removes fewer names from the global scope or the containing namespace.
- Nested classes allow extra structure to be expressed in the grammar of the language. For example, the name of the class is **Bank . Account** (three tokens) rather than **BankAccount** (one token).

Accessing Nested Classes



- Nested Classes Can Also Be Declared As Public or Private

```
class Bank
{
    public  class Account { ... }
    private class AccountNumberGenerator { ... }
}
class Program
{
    static void Main( )
    {
        Bank.Account                   accessible; ✓
        Bank.AccountNumberGenerator inaccessible; ✗
    }
}
```

You control the accessibility of data and methods by declaring them as public or private. You control the accessibility of a nested class in exactly the same way.

Public Nested Class

A public nested class has no access restrictions. It is declared to be publicly accessible. The full name of a nested class must still be used when outside the containing class.

Private Nested Class

A private nested class has exactly the same access restrictions as private data or methods. A private nested class is inaccessible from outside the containing class, as the following example shows:

```
class Bank
{
    private class AccountNumberGenerator( )
    {
        ...
    }
}
class Program
{
    static void Main( )
    {
        // Compile time error
        Bank.AccountNumberGenerator variable;
    }
}
```

In this example, **Main** cannot use **Bank.AccountNumberGenerator** because **Main** is a method of **Program** and **AccountNumberGenerator** is private and hence only accessible to its outer class, **Bank**.

A private nested class is accessible only to members of the containing class as the following examples shows:

```
class Bank
{
    public class Account
    {
        public void Setup( )
        {
            NumberSetter.Set(this);
            balance = 0M;
        }

        private class NumberSetter
        {
            public static void Set(Account a)
            {
                a.number = nextNumber++;
            }
            private static int nextNumber = 2311;
        }

        private int number;
        private decimal balance;
    }
}
```

In this code, note that the **Account.Setup** method can access the **NumberSetter** class because, although **NumberSetter** is a private class, it is private to **Account**, and **Setup** is a method of **Account**.

Notice also that the **Account.NumberSetter.Set** method can access the private *balance* field of the **Account** object *a*. This is because **Set** is a method of class **NumberSetter**, which is nested inside **Account**. Hence **NumberSetter** (and its methods) have access to the private members of **Account**.

The default accessibility of a nested class is private (as it is for data and methods). In the following example, the **Account** class defaults to private:

```
class Bank
{
    class Account( ) { ... }

    public Account OpenPublicAccount( )
    {
        Account opened = new Account( );
        opened.Setup( );
        return opened;
    }

    private Account OpenPrivateAccount( )
    {
        Account opened = new Account( );
        opened.Setup( );
        return opened;
    }
}
```

The **Account** class is accessible to **OpenPublicAccount** and **OpenPrivateAccount** because both methods are nested inside **Bank**. However, the **OpenPublicAccount** method will not compile. The problem is that **OpenPublicAccount** is a public method, usable as in the following code:

```
class Program
{
    static void Main( )
    {
        Bank b = new Bank( );
        Bank.Account opened = b.OpenPublicAccount( );
        ...
    }
}
```

This code will not compile because **Bank.Account** is not accessible to **Program.Main, Bank.Account** is private to **Bank,** and **Main** is not a method of **Bank**. The following error message appears:

```
error CS0050: Inconsistent accessibility: return type
'Bank.Account' is less accessible than method
'Bank.OpenPublicAccount'
```

The accessibility rules for a top-level class (that is, a class that is not nested inside another class) are not the same as those for a nested class. A top-level class cannot be declared private and defaults to internal accessibility. (Internal access is covered fully in a later module.)

Lab 7: Creating and Using Classes

Objectives

After completing this lab, you will be able to:

- Create classes and instantiate objects.
- Use non-static data and methods.
- Use static data and methods.

Prerequisites

Before working on this lab, you must be familiar with the following:

- Creating methods in C#
- Passing arguments as method parameters in C#

Estimated time to complete this lab: 45 minutes

Exercise 1
Creating and Using a Class

In this exercise, you will take the bank account struct that you developed in a previous module and convert it into a class. You will declare its data members as private but provide non-static public methods for accessing the data. You will build a test harness that creates an account object and populates it with an account number and balance that is specified by the user. Finally, you will print the data in the account.

↙ To change BankAccount from a struct to a class

1. Open the CreateAccount.sln project in the *install folder*\Labs\Lab07\Starter\CreateAccount folder.
2. Study the program in the BankAccount.cs file. Notice that *BankAccount* is a struct type.
3. Compile and run the program. You will be prompted to enter an account number and an initial balance. Repeat this process to create another account.
4. Modify *BankAccount* in BankAccount.cs to make it a class rather than a struct.
5. Compile the program. It will fail to compile. Open the CreateAccount.cs file and view the **CreateAccount** class. The class will look as follows:

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created;
        ...
        created.accNo = number; // Error here
        ...
    }
    ...
}
```

6. The assignment to *created.accNo* compiled without error when **BankAccount** was a struct. Now that it is a class, it does not compile! This is because when **BankAccount** was a struct, the declaration of the *created* variable created a **BankAccount** *value* (on the stack). Now that **BankAccount** is a class, the declaration of the *created* variable does not create a **BankAccount** value; it creates a **BankAccount** *reference* that does not yet refer to a **BankAccount** *object*.

7. Change the declaration of *created* so that it is initialized with a newly created **BankAccount** object, as shown:

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );
        ...
        created.accNo = number;
        ...
    }
    ...
}
```

8. Save your work.
9. Compile and run the program. Verify that the data entered at the console is correctly read back and displayed in the **CreateAccount.Write** method.

To encapsulate the BankAccount class

1. All the data members of the **BankAccount** class are currently public. Modify them to make them private, as shown:

```
class BankAccount
{
    private long accNo;
    private decimal accBal;
    private AccountType accType;
}
```

2. Compile the program. It will fail to compile. The error occurs in the **CreateAccount** class as shown::

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );
        ...
        created.accNo = number; // Error here again
        ...

    }
    ...
}
```

3. The **BankAccount** data member assignments now fail to compile because the data members are private. Only **BankAccount** methods can access the private **BankAccount** data members. You need to write a public **BankAccount** method to do the assignments for you. Perform the following steps:

   Add a non-static public method called **Populate** to **BankAccount**. This method will return **void** and expect two parameters: a long (the bank account number) and a decimal (the bank account balance). The body of this method will assign the long parameter to the *accNo* field and the decimal parameter to the *accBal* field. It will also set the *accType* field to **AccountType.Checking** as shown:

```
class BankAccount
{
    public void Populate(long number, decimal balance)
    {
        accNo = number;
        accBal = balance;
        accType = AccountType.Checking;
    }

    private long accNo;
    private decimal accBal;
    private AccountType accType;
}
```

4. Comment out the three assignments to the *created* variable in the **CreateAccount.NewbankAccount** method. In their place, add a statement that calls the **Populate** method on the *created* variable, passing **number** and **balance** as arguments. This will look as follows:

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );
        ...
        // created.accNo = number;
        // created.accBal = balance;
        // created.accType = AccountType.Checking;

        created.Populate(number, balance);
        ...
    }
    ...
}
```

5. Save your work.

6. Compile the program. It will fail to compile. There are still three statements in the **CreateAccount.Write** method that attempt to directly access the private **BankAccount** fields. You need to write three public **BankAccount** methods that return the values of these three fields. Perform the following steps:

   a. Add a non-static public method to **BankAccount** called **Number**. This method will return a long and expect no parameters. It will return the value of the *accNo* field as shown:

   ```
   class BankAccount
   {
       public void Populate(...) ...

       public long Number( )
       {
           return accNo;
       }
       ...
   }
   ```

   b. Add a non-static public method to **BankAccount** called **Balance**, as shown in the following code. This method will return a decimal and expect no parameters. It will return the value of the *accBal* field.

   ```
   class BankAccount
   {
       public void Populate(...) ...

       ...
       public decimal Balance( )
       {
           return accBal;
       }
       ...
   }
   ```

   c. Add a non-static public method called **Type** to **BankAccount**, as shown in the following code. This method will return an **AccountType** and expect no parameters. It will return the value of the *accType* field.

   ```
   class BankAccount
   {
       public void Populate(...) ...

       ...
       public AccountType Type( )
       {
           return accType;
       }
       ...
   }
   ```

d. Finally, replace the three statements in the **CreateAccount.Write** method that attempt to directly access the private **BankAccount** fields with calls to the three public methods you have just created, as shown:

```
class CreateAccount
{
    ...
    static void Write(BankAccount toWrite)
    {
        Console.WriteLine("Account number is {0}",
➥toWrite.Number( ));
        Console.WriteLine("Account balance is {0}",
➥toWrite.Balance( ));
        Console.WriteLine("Account type is {0}",
➥toWrite.Type( ).Format( ));
    }
}
```

7. Save your work.
8. Compile the program and correct any other errors. Run the program. Verify that the data entered at the console and passed to the **BankAccount.Populate** method is correctly read back and displayed in the **CreateAccount.Write** method.

To further encapsulate the BankAccount class

1. Change the **BankAccount.Type** method so that it returns the type of the account as a string rather than as an **AccountType** enum, as shown:

```
class BankAccount
{
    ...
    public string Type( )
    {
        return accType.Format( );
    }
    ...
    private AccountType accType;
}
```

2. Change the last **WriteLine** statement in the **CreateAccount.Write** method so that it no longer calls the **Format** method, as shown:

```
class CreateAccount
{
    ...
    static void Write(BankAccount acc)
    {
        Console.WriteLine("Account number is {0}",
➥acc.Number( ));
        Console.WriteLine("Account balance is {0}",
➥acc.Balance( ));
        Console.WriteLine("Account type is {0}",
➥acc.Type( ));
    }
}
```

3. Save your work.
4. Compile the program and correct any errors. Run the program. Verify that the data entered at the console and passed to the **BankAccount.Populate** method is correctly read back and displayed in the **CreateAccount.Write** method.

Exercise 2
Generating Account Numbers

In this exercise, you will modify the **BankAccount** class from Exercise 1 so that it will generate unique account numbers. You will accomplish this by using a static variable in the **BankAccount** class and a method that increments and returns the value of this variable. When the test harness creates a new account, it will call this method to generate the account number. It will then call the method of the **BankAccount** class that sets the number for the account, passing in this value as a parameter.

✓ To ensure that each BankAccount number is unique

1. Open the project UniqueNumbers.sln in the *install folder*\Labs\Lab07\Starter\UniqueNumbers folder.

Note This project is the same as the completed CreateAccount project from Exercise 1.

2. Add a private static long called *nextAccNo* to the **BankAccount** class, as shown:

```
class BankAccount
{
    ...
    private long accNo;
    private decimal accBal;
    private AccountType accType;

    private static long nextAccNo;
}
```

3. Add a public static method called **NextNumber** to the **BankAccount** class, as shown in the following code. This method will return a long and expect no parameters. It will return the value of the *nextAccNo* field in addition to incrementing this field.

```
class BankAccount
{
    ...
    public static long NextNumber( )
    {
        return nextAccNo++;
    }

    private long accNo;
    private decimal accBal;
    private AccountType accType;

    private static long nextAccNo;
}
```

4. Comment out the statement in the **CreateAccount.NewBankAccount** method that writes a prompt to the console asking for the bank account number, as shown:

   ```
   //Console.Write("Enter the account number: ");
   ```

5. Replace the initialization of *number* in the **CreateAccount.NewBankAccount** method with a call to the **BankAccount.NextNumber** method you have just created, as shown:

   ```
   //long number = long.Parse(Console.ReadLine( ));
   long number = BankAccount.NextNumber( );
   ```

6. Save your work.
7. Compile the program and correct any errors. Run the program. Verify that the two accounts have account numbers 0 and 1.
8. Currently, the **BankAccount.nextAccNo** static field has a default initialization to zero. Explicitly initialize this field to 123.
9. Compile and run the program. Verify that the two accounts created have account numbers 123 and 124.

To further encapsulate the BankAccount class

1. Change the **BankAccount.Populate** method so that it expects only one parameter—the decimal *balance*. Inside the method, assign the *accNo* field by using the **BankAccount.NextNumber** static method, as shown:

```
class BankAccount
{
    public void Populate(decimal balance)
    {
        accNo = NextNumber( );
        accBal = balance;
        accType = AccountType.Checking;
    }
    ...
}
```

2. Change **BankAccount.NextNumber** into a private method, as shown:

```
class BankAccount
{
    ...
    private static long NextNumber( ) ...
}
```

3. Comment out the declaration and initialization of *number* in the **CreateAccount.NewBankAccount** method. Change the **created.Populate** method call so that it only passes a single parameter, as shown:

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );

        //long number = BankAccount.NextNumber( );
        ...
        created.Populate(balance);
        ...
    }
    ...
}
```

4. Save your work.
5. Compile the program and correct any errors. Run the program. Verify that the two accounts still have account numbers 123 and 124.

Exercise 3
Adding More Public Methods

In this exercise, you will add two methods to the **Account** class: **Withdraw** and **Deposit**.

Withdraw will take a decimal parameter and will deduct the given amount from the balance. However, it will check first to ensure that sufficient funds are available, since accounts are not allowed to become overdrawn. It will return a bool value indicating whether the withdrawal was successful.

Deposit will also take a decimal parameter whose value it will add to the balance in the account. It will return the new value of the balance.

↙ To add a Deposit method to the BankAccount class

1. Open the project MoreMethods.sln in the *install folder*\ Labs\Lab07\Starter\MoreMethods folder.

   Note This project is the same as the completed UniqueNumbers project from Exercise 2.

2. Add a public non-static method called **Deposit** to the **BankAccount** class, as shown in the following code. This method will also take a decimal parameter whose value it will add to the balance in the account. It will return the new value of the balance.

```
class BankAccount
{
    ...
    public decimal Deposit(decimal amount)
    {
        accBal += amount;
        return accBal;
    }
    ...
}
```

3. Add a public static method called **TestDeposit** to the **CreateAccount** class, as shown in the following code. This method will return **void** and expect a **BankAccount** parameter. The method will write a prompt to the console prompting the user for the amount to deposit, capture the entered amount as a decimal, and then call the **Deposit** method on the **BankAccount** parameter, passing the amount as an argument.

```
class CreateAccount
{
    ...
    public static void TestDeposit(BankAccount acc)
    {
        Console.Write("Enter amount to deposit: ");
        decimal amount = decimal.Parse(Console.ReadLine());
        acc.Deposit(amount);
    }
    ...
}
```

4. Add to **CreateAccount.Main** statements that call the **TestDeposit** method you have just created, as shown in the following code. Ensure that you call **TestDeposit** for both account objects. Use the **CreateAccount.Write** method to display the account after the deposit takes place.

```
class CreateAccount
{
    static void Main( )
    {
        BankAccount berts = NewBankAccount( );
        Write(berts);
        TestDeposit(berts);
        Write(berts);

        BankAccount freds = NewBankAccount( );
        Write(freds);
        TestDeposit(freds);
        Write(freds);
    }
}
```

5. Save your work.
6. Compile the program and correct any errors. Run the program. Verify that deposits work as expected.

Note If you have time, you might want to add a further check to **Deposit** to ensure that the decimal parameter passed in is not negative.

✓ To add a Withdraw method to the BankAccount class

1. Add a public non-static method called **Withdraw** to **BankAccount**, as shown in the following code. This method will expect a decimal parameter specifying the amount to withdraw. It will deduct the amount from the balance only if sufficient funds are available, since accounts are not allowed to become overdrawn. It will return a bool indicating whether the withdrawal was successful.

```
class BankAccount
{
    ...
    public bool Withdraw(decimal amount)
    {
        bool sufficientFunds = accBal >= amount;
        if (sufficientFunds) {
            accBal -= amount;
        }
        return sufficientFunds;
    }
    ...
}
```

2. Add a public static method called **TestWithdraw** to the **CreateAccount** class, as shown in the following code. This method will return **void** and will expect a **BankAccount** parameter. The method will write a prompt to the console prompting the user for the amount to withdraw, capture the entered amount as a decimal, and then call the **Withdraw** method on the **BankAccount** parameter, passing the amount as an argument. The method will capture the bool result returned by **Withdraw** and write a message to the console if the withdrawal failed.

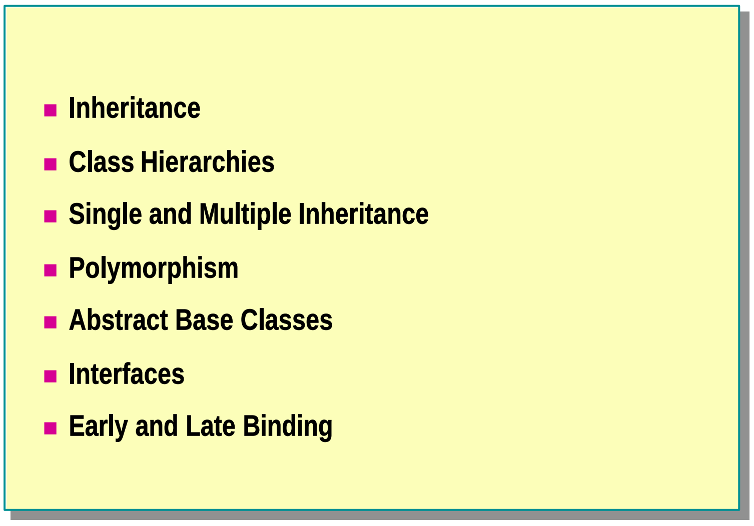
```
class CreateAccount
{
    ...
    public static void TestWithdraw(BankAccount acc)
    {
        Console.Write("Enter amount to withdraw: ");
        decimal amount = decimal.Parse(Console.ReadLine());
        if (!acc.Withdraw(amount)) {
            Console.WriteLine("Insufficient funds.");
        }
    }
    ...
}
```

3. Add to **CreateAccount.Main** statements that call the **TestWithdraw** method you have just created, as shown in the following code. Ensure that you call **TestWithdraw** for both account objects. Use the **CreateAccount.Write** method to display the account after the withdrawal takes place.

```
class CreateAccount
{
    static void Main( )
    {
        BankAccount berts = NewBankAccount( );
        Write(berts);
        TestDeposit(berts);
        Write(berts);
        TestWithdraw(berts);
        Write(berts);

        BankAccount freds = NewBankAccount( );
        Write(freds);
        TestDeposit(freds);
        Write(freds);
        TestWithdraw(freds);
        Write(freds);
    }
}
```

4. Save your work.
5. Compile the program and correct any errors. Run the program. Verify that withdrawals work as expected. Test successful and unsuccessful withdrawals.

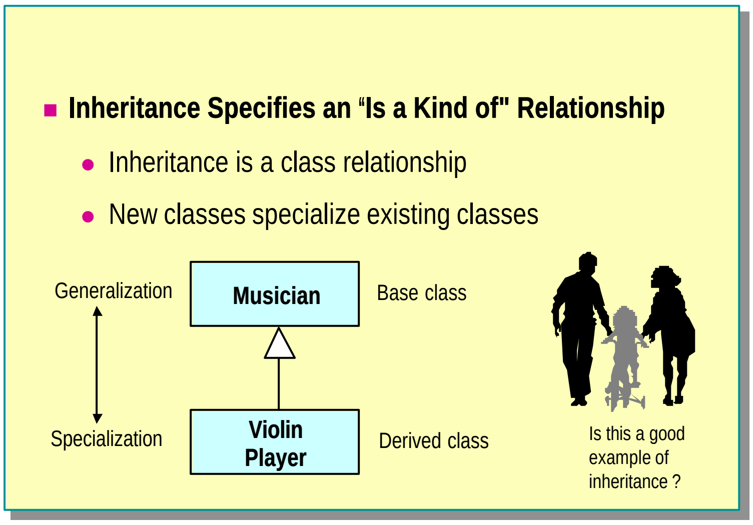
◆ Defining Object-Oriented Systems



- Inheritance
- Class Hierarchies
- Single and Multiple Inheritance
- Polymorphism
- Abstract Base Classes
- Interfaces
- Early and Late Binding

In this section, you will learn about inheritance and polymorphism. You will learn how to implement these concepts in C# in later modules.

Inheritance



- **Inheritance Specifies an "Is a Kind of" Relationship**
  - Inheritance is a class relationship
  - New classes specialize existing classes

Inheritance is a relationship that is specified at the class level. A new class can be derived from an existing class. In the slide above, the **ViolinPlayer** class is derived from the **Musician** class. The **Musician** class is called the *base* class (or, less frequently, the parent class, or the superclass); the **ViolinPlayer** class is called the *derived* class (or, less frequently, the child class, or subclass). The inheritance is shown by using the Unified Modeling Language (UML) notation. More UML notation will be covered in later slides.

Inheritance is a powerful relationship because a derived class inherits everything from its base class. For example, if the base class **Musician** contains a method called **TuneYourInstrument**, this method is automatically a member of the derived **ViolinPlayer** class.

A base class can have any number of derived classes. For example, new classes (such as **FlutePlayer**, or **PianoPlayer)** could all be derived from the **Musician** class. These new derived classes would again automatically inherit the **TuneYourInstrument** method from the **Musician** base class.

Note A change to a base class is automatically a change to all derived classes. For example, if a field of type **MusicalIntrument** was added to the **Musician** base class, then every derived class (**ViolinPlayer**, **FlutePlayer**, **PianoPlayer**, and so on) would automatically acquire a field of type **MusicalInstrument**. If a bug is introduced into a base class, it will automatically become a bug in every derived class. (This is known as the *fragile base class problem*.)

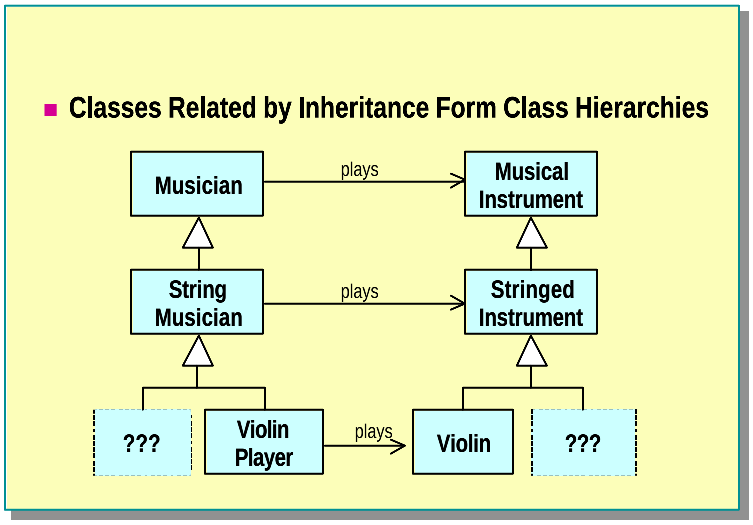
Understanding Inheritance in Object-Oriented Programming

The graphic on the slide shows a man, a woman, and a small girl riding a bicycle. If the man and the woman are the biological parents of the girl, then she will inherit half of her genes from the man and half of her genes from the woman.

But this is not an example of class inheritance. It is implementation mechanism!

The classes are **Man** and **Woman**. There are two instances of the **Woman** class (one with an **age** attribute of less than 16) and one instance of the **Man** class. There is no class inheritance. The only possible way there could be class inheritance in this example is if the **Man** class and the **Woman** class share a base class **Person**.

Class Hierarchies



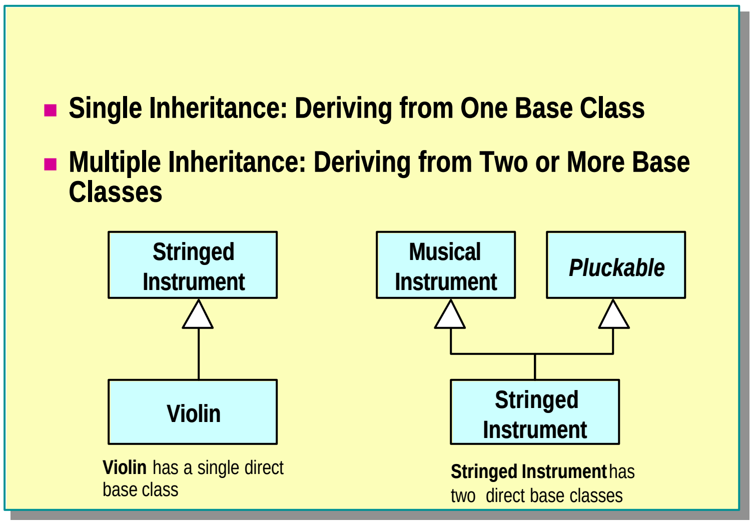
- Classes Related by Inheritance Form Class Hierarchies

Classes that derive from base classes can themselves be derived from. For example, in the slide the **StringMusician** class is derived from the **Musician** class but is itself a base class for the further derived **ViolinPlayer** class. A group of classes related by inheritance forms a structure known as a *class hierarchy*. As you move up a hierarchy, the classes represent more general concepts (generalization); as you move down a hierarchy the classes represent more specialized concepts (specialization).

The depth of a class hierarchy is the number of levels of inheritance in the hierarchy. Deeper class hierarchies are harder to use and harder to implement than shallow class hierarchies. Most programming guidelines recommend that the depth be limited to between five and seven classes.

The slide depicts two parallel class hierarchies: one for musicians and another for musical instruments. Creating class hierarchies is not easy: classes need to be designed as base classes from the start. Inheritance hierarchies are also the dominant feature of frameworks—models of work that can be built on and extended.

Single and Multiple Inheritance



- Single Inheritance: Deriving from One Base Class
- Multiple Inheritance: Deriving from Two or More Base Classes

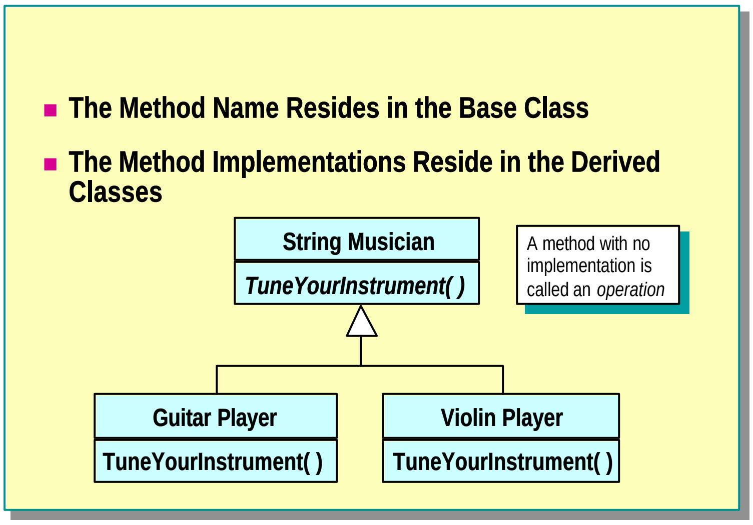
Stringed Instrument
Violin
Violin has a single direct base class
Musical Instrument
Pluckable
Stringed Instrument
Stringed Instrument has two direct base classes

Single inheritance occurs when a class has a single direct base class. In the example in the slide, the **Violin** class inherits from one class, **StringedInstrument**, and is an example of single inheritance. **StringedInstrument** derives from two classes, but that is not relevant to the **Violin** class. Single inheritance can still be difficult to use wisely. It is well known that inheritance is one of the most powerful software modeling tools, and at the same time one of the most misunderstood and misused.

Multiple inheritance occurs when a class has two or more direct base classes. In the example in the slide, the **StringedInstrument** class derives directly from two classes, **MusicalInstrument** and **Pluckable**, and provides an example of multiple inheritance. Multiple inheritance offers multiple opportunities to misuse inheritance! C#, like most modern programming languages (but not C++), restricts the use of multiple inheritance: you can inherit from as many interfaces as you want, but you can only inherit from one non-interface (that is, at most one abstract or concrete class). The terms interface, abstract class, and concrete class are covered later in this module.

Notice that all forms of inheritance, but multiple inheritance in particular, offer many views of the same object. For example, a **Violin** object could be used at the **Violin** class level, but it could also be used at the **StringedInstrument** class level.

Polymorphism



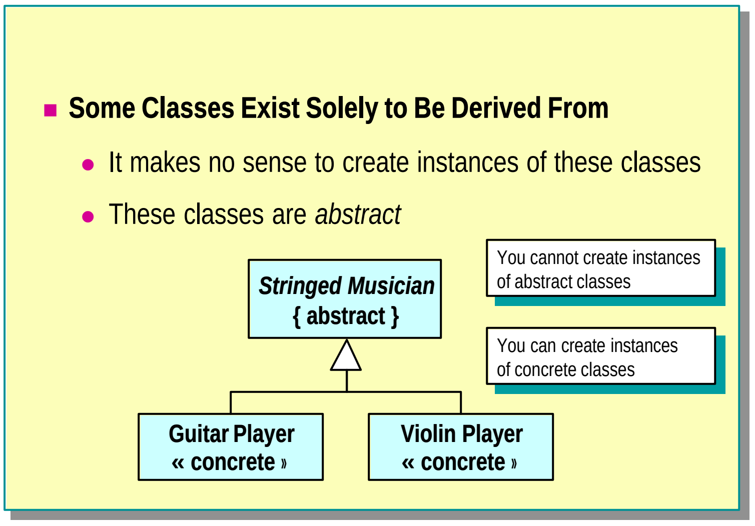
- The Method Name Resides in the Base Class
- The Method Implementations Reside in the Derived Classes

Polymorphism literally means *many forms* or *many shapes*. It is the concept that a method declared in a base class can be implemented in many different ways in the different derived classes.

Consider the scenario of an orchestra of musicians all tuning their instruments as they get ready for a concert. Without polymorphism, the conductor needs to visit each musician in turn, seeing what kind of instrument the musician plays, and giving detailed instructions about how to tune that particular kind of instrument. With polymorphism, the conductor just tells each musician, “tune your instrument.” The conductor does not need to know which particular instrument each musician plays, just that each musician will respond to the same request for behavior in a manner appropriate to their particular instrument. Rather than the conductor being responsible for the knowledge of how to tune all of the different kinds of instruments, the knowledge is partitioned across the different kinds of musicians as appropriate: a guitar player knows how to tune a guitar, a violin player knows how to tune a violin. In fact, the conductor does not know how to tune *any* of the instruments. This decentralized allocation of responsibilities also means that new derived classes (such as **DrumPlayer**) can be added to the hierarchy without necessarily needing to modify existing classes (such as the conductor).

There is one problem though. What is the body of the method at the base-class level? Without knowing which particular kind of instrument a musician plays, it is impossible to know how to tune the instrument. To manage this, only the name of the method (and no body) can be declared in the base class. A method name with no method body is called an *operation*. One of the ways of denoting an operation in UML is to use italics, as is shown in the slide.

Abstract Base Classes



- **Some Classes Exist Solely to Be Derived From**
  - It makes no sense to create instances of these classes
  - These classes are *abstract*

Stringed Musician
{ abstract }

Guitar Player
« concrete »

Violin Player
« concrete »

You cannot create instances of abstract classes

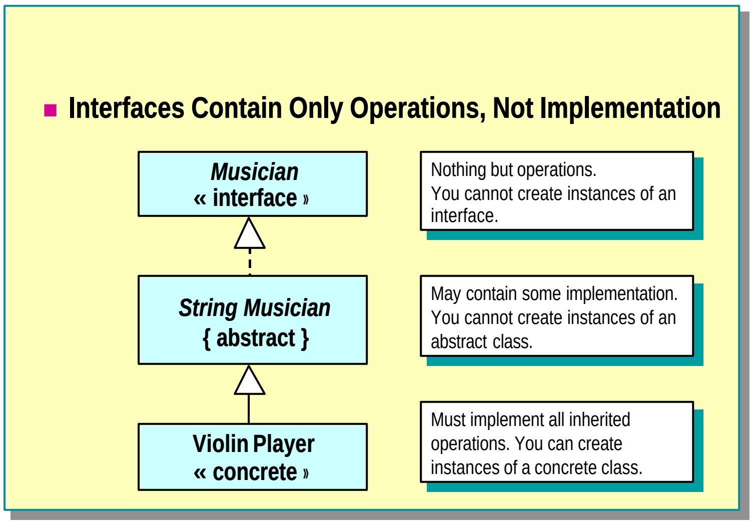
You can create instances of concrete classes

In a typical class hierarchy, the operation (the name of a method) is declared in the base class, and the method is implemented in different ways in the different derived classes. The base class exists solely to introduce the name of the method into the hierarchy. In particular, the base class operation does not require an implementation. This makes it vital that the base class not be used as a regular class. Most importantly, you must not be allowed to create instances of the base class: if you could, what would happen if you called the operation that had no implementation? A mechanism is required that makes it impossible to create instances of these base classes: the base class needs to be marked abstract.

In a UML design, you can constrain a class as abstract by writing the name of the class in italics or by placing the word *abstract* within braces ({ and }). In contrast, you can use the word *concrete* or *class* between guillemets (<< and >>) as a stereotype to denote in UML a class that is not abstract, a class that can be used to create instances. This is shown in the slide. All object-oriented programming languages have grammatical constructs that implement an abstract constraint. (Even C++ can use protected constructors.)

Sometimes the creation of an abstract base class is more retrospective: duplicate common features in the derived classes are factored into a new base class. However, once again, the base class should be marked abstract because its purpose is to be derived from, and not to create instances.

Interfaces



- Interfaces Contain Only Operations, Not Implementation

Musician
« interface »

Nothing but operations.
You cannot create instances of an interface.

String Musician
{ abstract }

May contain some implementation.
You cannot create instances of an abstract class.

Violin Player
« concrete »

Must implement all inherited operations. You can create instances of a concrete class.

Abstract classes and interfaces are alike in that neither can be used to instantiate objects. However, they differ in that an abstract class may contain some implementation whereas an interface contains no implementation of any kind; an interface contains only operations (the names of methods). You could say that an interface is even more abstract than an abstract class!

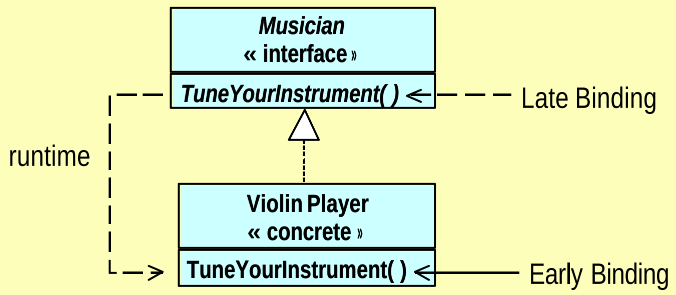
In UML, you can depict an interface by using the word *interface* between guillemets (<< and >>). All object-oriented programming languages have grammatical constructs that implement an interface.

Interfaces are important constructs in object-oriented programs. In UML, interfaces have specific notation and terminology. When you derive from an interface, it is said that you *implement* that interface. UML depicts this with a dashed line called *realization*. When you derive from a non-interface (an abstract class or a concrete class) it is said that you *extend* that class. UML depicts this with a solid line called *generalization/specialization*.

Place your interfaces at the top of a class hierarchy. The idea is simple: if you can program to an interface—that is, if you use only those features of an object that are declared in its interface—your program loses all dependence on the specific object and its concrete class. In other words, when you program to an interface, many different objects of many different classes can be used interchangeably. It is this ability to make changes with no impact that leads to the object-oriented maxim, "Program to an interface and not to an implementation."

Early and Late Binding

- Normal Method Calls Are Resolved at Compile Time
- Polymorphic Method Calls Are Resolved at Run Time



When you make a method call directly on an object, that is, not through a base class operation, the method call is resolved at compile time. This is also known as *early binding* or *static binding*.

When you make a method call indirectly on an object—that is, through a base class operation—the method call is resolved at run time. This is also known as *late binding* or *dynamic binding*.

An example of late binding occurs when a conductor tells all of the musicians in an orchestra to tune their instruments. By working at the interface level, the conductor does not need to know (and hence be dependent on) the specific different kinds of concrete musicians (such as **ViolinPlayer**). The conductor is also freed from needing to know when a new class is added to the hierarchy for a new kind of musician (for example, **HarpPlayer**).

The flexibility of late binding comes with a physical price and a logical price:

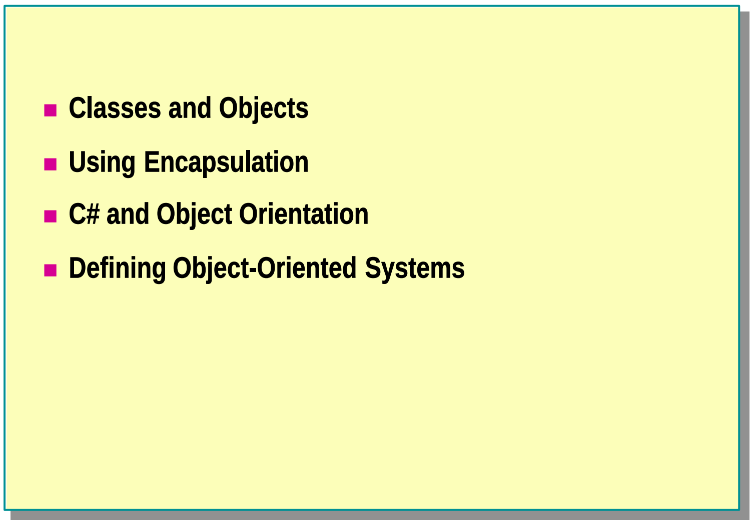
- Physical price

  Late bound calls are slightly slower than early bound calls. In effect, the extra work that must be performed as a result of a late bound call is to discover the class of the calling object. This is done in an efficient manner (you would not be able to do it faster yourself), but it is extra work.

- Logical price

  With late binding, derived classes can be substituted for their base classes. An operation call can be made through an interface, and at run time the derived class object will correctly have its method called. In other words, all derived classes that implement an interface can act as substitutes for the interface type. Newcomers to object-oriented programming often fail to fully appreciate the substitutability aspect of inheritance.

Review



- Classes and Objects
- Using Encapsulation
- C# and Object Orientation
- Defining Object-Oriented Systems

1. Explain the concept of abstraction and why it is important in software engineering.

2. What are the two principles of encapsulation?

3. Describe inheritance in the context of object-oriented programming.

4. What is polymorphism? How is it related to early and late binding?

5. Describe the differences between interfaces, abstract classes, and concrete classes.