

Module 3: Creating Objects in C#

Contents

Overview	1
Lesson: Defining a Class	2
Lesson: Declaring Methods	18
Lesson: Using Constructors	35
Lesson: Using Static Class Members	44
Review	52
Lab 3.1: Creating Classes in C#	54



Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2002 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, FrontPage, IntelliSense, JScript, Microsoft Press, MSDN, PowerPoint, Visual Basic, Visual C++, Visual C#, Visual Studio, Win32, Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Overview

- **Defining a Class**
- **Declaring Methods**
- **Using Constructors**
- **Using Static Class Members**

Introduction

This module introduces the fundamentals of object-oriented programming, including the concepts of objects, classes, and methods. It explains how to define classes and create objects, how to organize classes by using namespaces, and how to define, write, and call methods. Finally, it describes how to use constructors.

Objectives

After completing this module, you will be able to:

- Define a class
- Declare methods
- Use constructors
- Use static class members

Lesson: Defining a Class

- What Are Classes and Objects?
- What Are Value Types and Reference Types?
- How to Define a Class and Create an Object
- How to Organize Classes Using Namespaces
- How to Define Accessibility and Scope

Introduction

This lesson discusses how to define classes, instantiate objects, access class members, and use namespaces to organize classes.

Lesson objectives

After completing this lesson, you will be able to:

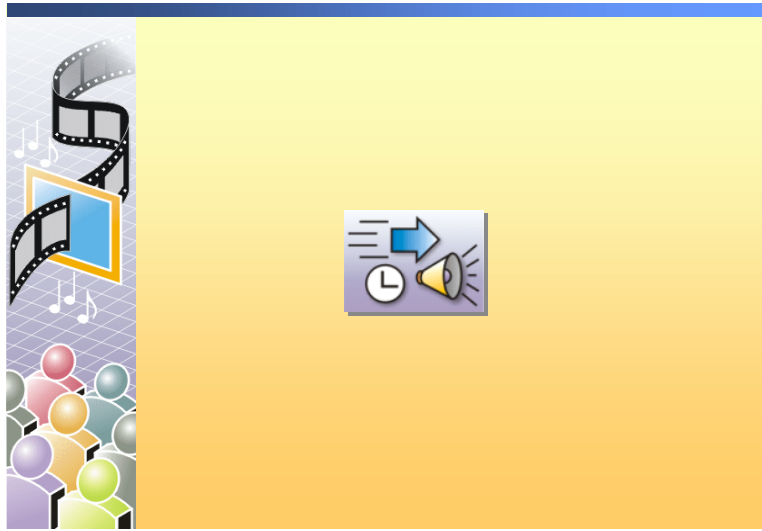
- Define a class.
- Create an object.
- Use access modifiers to define the scope of class members.
- Organize classes by using namespaces.

Lesson agenda

This lesson includes the following topics and activities:

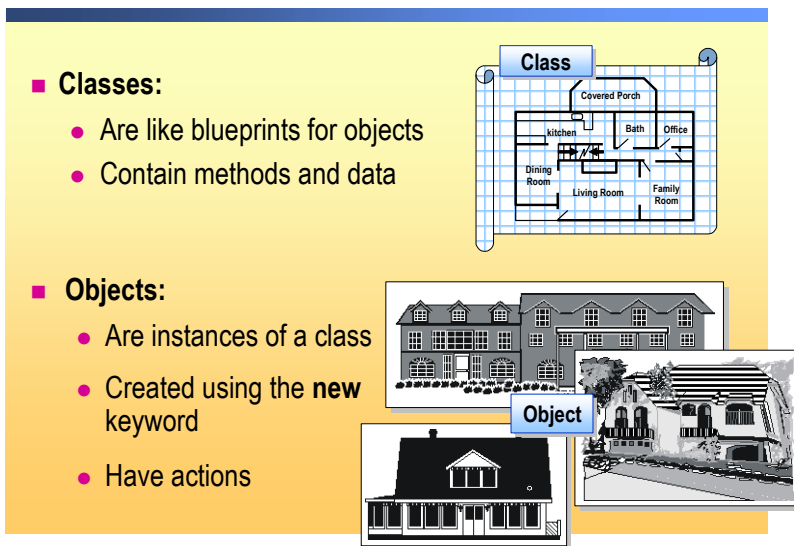
- Multimedia: Introduction to Classes and Objects
- What Are Classes and Objects?
- What Are Value Types and Reference Types?
- How to Define a Class and Create an Object
- How to Organize Classes Using Namespaces
- How to Define Accessibility and Scope
- Practice: Defining Classes and Creating Objects

Multimedia: What Are Objects and Classes?



This animation introduces you to the fundamental user-defined type in C#, the class.

What Are Classes and Objects?



Introduction

A class is the fundamental user-defined type in C#. You must define a class before you can create an object.

Definition

A *class* is essentially like a blueprint, from which you can create objects. A class defines the characteristics of an object, including properties that define the types of data that the object can contain and methods that describe the behavior of the object. These characteristics determine how other objects can access and work with the data that is contained in the object.

An *object* is an instance of a class. If a class is like a blueprint, then an object is what is created from that blueprint. The class is the *definition* of an item; the object *is* the item. The blueprint for your house is like a class; the house that you live in is an object.

Example 1

For example, if you want to build objects that represent ducks, you can define a **Duck** class that has certain behaviors, such as walking, quacking, flying, and swimming—and specific properties, such as height, weight, and color. It is important to notice that the behaviors are relevant to the object. Although it is obviously illogical to create a duck object that barks like a dog, relating behavior to objects is not always so clear when you work with the type of data that a programmer typically manipulates.

The **Duck** *class* defines what a duck is and what it can do. A **Duck** *object* is a specific duck that has a specific weight, color, height, and behavioral characteristics. The duck that you feed is a duck object.

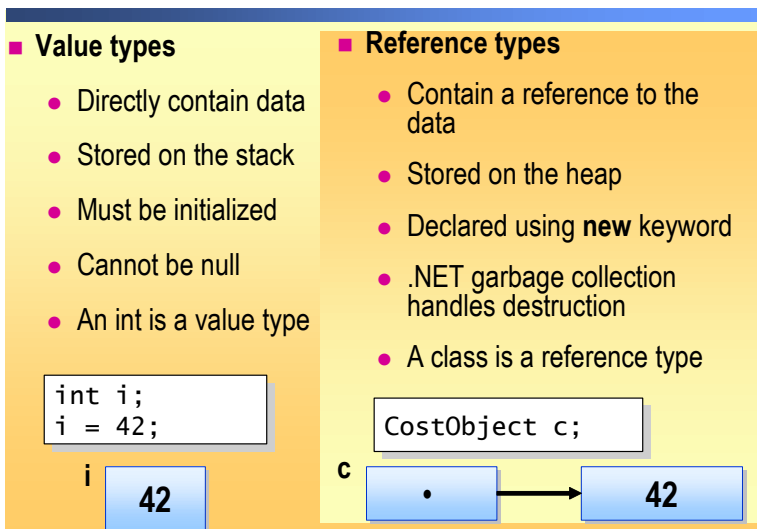
Example 2

Suppose that a programmer must write a function that changes a customer's address in a database. In a traditional approach, the programmer may write a **ChangeAddress** function that takes a database table and row as a parameter and changes the address information in that row. Or, the programmer may use the person's name as a parameter, search the table for that name, and then change the address in the record. The disadvantage of this approach is that when you want to change the information about the person, you must know something about how that information is represented, in this case, in a specific table in a database.

An object-oriented approach is to define a class that represents customers and provides the ability to change addresses. The application that uses the **Customer** class is likely to manage multiple customer objects, each representing one customer. Each customer object contains information about the location of that customer's record in the database, so when the application must change the address information, it can simply invoke the **ChangeAddress** action or method for that particular customer. The application tells the **Customer** object to change its address, in effect.

In C#, everything behaves like an object. When you create an object, you are creating a new type, called a reference type.

What Are Value Types and Reference Types?



Introduction

There are two kinds of types: value types and reference types. Most of the predefined types are value types. For example, an integer is a value type.

Value types

Value types directly contain their data. Therefore, each value type variable directly contains the value that it is assigned.

Value types store themselves, and the data that they contain, in an area of memory called the *stack*. The stack is an area of memory that is used to store items in a last-in, first-out manner.

Reference types

Reference type variables contain a reference to their data. Objects are reference types.

More than one reference type variable can reference the same object. Therefore, it is possible for operations on one reference type variable to affect other variables that refer to the same object, the same data. Reference types contain a reference to data that is allocated on the *heap*. The heap is an area of memory where objects are allocated.

Initializing a value type

When you declare a value type variable, you must then initialize it before it can be used. To initialize a value type variable, you simply assign a value to that variable, as shown in the following example:

```
int anInteger;
anInteger = 42;
```

The first line declares the value type variable **int** by naming it **anInteger** and the second line initializes the variable by assigning it a value of **42**.

Initializing a reference type

When you declare a reference type variable, you then initialize it by using the **new** keyword. This keyword allocates some memory on the heap for your variable. For example, suppose that you have a class named **Customer**.

```
Customer bestCustomer = new Customer();
```

The variable **bestCustomer** refers to an object of type **Customer**.

Boxing

C# allows you to treat value types like reference types. For example, you can declare an integer, assign a value to it, and call the **ToString** method.

Console.WriteLine method uses *boxing* in exactly this manner, to write the string format of the parameters that are passed to it.

```
int x = 25;  
Console.WriteLine( x );
```

`x` is declared as a value type, but when the **ToString** method is invoked, it is converted to an object, which is what provides the **ToString** method. This process is called *boxing*. Boxing occurs implicitly when you use a value type where a reference type is expected. You can also explicitly box a value type by converting it to an object type, as shown in the following example:

```
object boxedValue = (object) x;
```

You can convert `x` back to a value type, although you must do so explicitly. This is called *unboxing*. To do this, simply copy the boxed variable to a value type variable, explicitly converting it to the correct value type, as shown in the following example:

```
int y = (int) boxedValue;
```

You can unbox a value only if it can be assigned to the unboxed value type.

How to Define a Class and Create an Object

■ How to define a class

```
public class Customer {
    public string    name;
    public decimal   creditLimit;
    public uint      customerID;
}
```

■ How to instantiate a class as an object

```
Customer nextCustomer = new Customer();
```

■ How to access class variables

```
nextCustomer.name = "Suzan Fine";
```

Introduction

A class is like a blueprint that is used to create objects, in the same way that a blueprint for a house is used to create many individual houses.

Syntax

To define a class, you place the **class** keyword before the name of your class, and then you insert the class members between braces.

```
[attributes] [access-modifiers] class identifier {class-body}
```

Pascal case

It is recommended that you use Pascal case for your class name, for example, **MyClassName**. Pascal case means that the initial letter of each word in the identifier is capitalized.

Example

The following example defines a new class, **Customer**, with three associated pieces of relevant information—the customer's name, the credit limit of the customer, and a customer ID. Although the **Customer** class is defined in the example, there are no **Customer** objects yet. They still must be created.

```
class Customer {
    public string name;
    public decimal creditLimit;
    public uint   customerID;
}
```

A class is a user-defined type, as opposed to a system-provided type. When you define a class, you actually create a new type in your application. To use a class that you have defined, you must first instantiate an object of that type by using the **new** keyword.

Syntax

```
<class> <object> = new <class>
```

```
Customer nextCustomer = new Customer();
```

Accessing class variables

After you instantiate an object, to access and use the data that the object contains, you type the name of the instantiated class, followed by a period and the name of the class member that you want to access.

For example, you can access the **name** member of the **Customer** class and assign it a value in your **nextCustomer** object, the name of the instantiated class, as follows:

```
nextCustomer.name = "Suzan Fine";
```

Example 2

The following code defines a new class named **Lion** with one class member, **weight**, and creates an instance of the **Lion** class, an object named **zooLion**. A value is assigned to the **weight** member of the **zooLion** class.

```
public class Lion {  
    public int weight;  
}  
  
. . .  
  
Lion zooLion = new Lion();  
zooLion.weight = 200;
```

Classes are reference types

Each **Lion** object that is created is a separate object, as shown in the following code:

```
Lion largerLion = new Lion();  
Lion smallerLion = new Lion();  
largerLion.weight = 200;
```

The preceding code does not change the **weight** member of **smallerLion** object. That value is zero, which is the default value for an integer.

The following code does change the **weight** member of the **smallerLion** object:

```
Lion largerLion = new Lion();  
largerLion.weight = 225;  
  
Lion smallerLion = new Lion();  
smallerLion.weight = 175;  
  
Lion recentlyWeighedLion = smallerLion;  
  
recentlyWeighedLion.weight = 185;  
  
// smallerLion's weight is now 185.
```

In the preceding code, the value of **smallerLion.weight** is **185**.

Because **Lion** is a reference type, the assignment to **recentlyWeighedLion** causes **smallerLion** and **recentlyWeighedLion** to reference the same object.

Object destruction

When you create an object, you are actually allocating some space in memory for that object. The Microsoft® .NET Framework provides an automatic memory management feature called *garbage collection*.

Garbage collection

The garbage collector monitors the lifetime of objects and frees the memory that is allocated to them when they are no longer being referenced. By working in the .NET Framework, a programmer no longer needs to worry about de-allocation and destruction of objects in memory.

When the garbage collector locates objects that are no longer being referenced, it implicitly executes the termination code that de-allocates the memory and returns it to the pool.

The garbage collector does not operate on a predictable schedule. It can run at unpredictable intervals, usually whenever memory becomes low.

Occasionally, you may want to dispose of your objects in a deterministic manner. For example, if you must release scarce resources over which there may be contention, such as terminating a database connection or a communication port, you can do so by using the **IDisposable** interface.

Note For more information about interfaces, see Module 5, “Programming with C#,” in Course 2609, *Introduction to C# Programming with Microsoft .NET*.

How to Organize Classes Using Namespaces

■ Declaring a namespace

```
namespace CompanyName {  
    public class Customer () { }  
}
```

■ Nested namespaces

```
namespace CompanyName {  
    namespace Sales {  
        public class Customer () { }  
    }  
}  
// Or  
namespace CompanyName.Sales { ... }
```

■ The using statement

```
using System;  
using CompanyName.Sales;
```

Introduction

You use namespaces to organize classes into a logically related hierarchy. Namespaces function as both an internal system for organizing your application and as an external way to avoid name clashes (collisions) between your code and other applications.

Because more than one company may create classes with the same name, such as “Customer,” when you create code that may be seen or used by third parties, it is highly recommended that you organize your classes by using a hierarchy of namespaces. This practice enables you to avoid interoperability issues.

Definition

A *namespace* is an organizational system that is used to identify groups of related classes.

Creating a namespace

To create a namespace, you simply type the keyword **namespace** followed by a name.

Best practices

It is recommended that you use Pascal case for namespaces.

It is also recommended that you create at least a two-tiered namespace, which is one that contains two levels of classification, separated by a period. Typically, you use your company name, followed by the name of a department or a product line.

Example

The following code shows an example of a two-tiered namespace:

```
namespace CompanyName.Sales {  
    // define your classes within this namespace  
    public class Customer() {  
  
    }  
}
```

The preceding two-tiered namespace declaration is identical to writing each namespace in a nested format, as shown in the following code:

```
namespace CompanyName {  
    namespace Sales {  
        public class Customer() {  
  
        }  
    }  
}
```

In both cases, you can refer to the class by using the following code:

```
CompanyName.Sales.Customer()
```

This is the *fully qualified name* of the **Customer** class. Users of the **Customer** class can use the fully qualified name to refer to this specific customer class and avoid name collisions with other **Customer** classes.

Note You should avoid creating a class with the same name as a namespace.

Commonly used namespaces in the .NET Framework

The Microsoft .NET Framework is made up of many namespaces, the most important of which is named **System**. The **System** namespace contains the classes that most applications use to interact with the operating system.

For example, the **System** namespace contains the **Console** class, which provides several methods, including **WriteLine**, which is a command that enables you to write code to an on-screen console. You can access the **WriteLine** method of the **Console** class as follows:

```
System.Console.WriteLine("Hello, World");
```

A few of the other namespaces that are provided by the .NET Framework through the **System** namespace are listed in the following table.

Namespace	Definition
System.Windows.Forms	Provides the classes that are useful for building applications based on Microsoft Windows®
System.IO	Provides classes for reading and writing data to files
System.Data	Provides classes that are useful for data access
System.Web	Provides classes that are useful for building Web Forms applications

The using directive

There is no limit to the number of tiers that a namespace can contain and, therefore, namespaces can grow long and cumbersome. To make code more readable, you can apply the **using** directive.

The **using** directive is a shortcut that tells your application that the types in the namespace can be referenced directly, without using the fully qualified name. Normally, at the top of your code file, you simply list the namespaces that you use in that file, prefixed with the **using** statement. You can put more than one **using** directive in the source file.

Example

```
using System;  
using CompanyName.Sales;  
...  
Console.WriteLine("Hello, World");  
Customer nextCustomer = new Customer();
```


How to Define Accessibility and Scope

- Access modifiers are used to define the accessibility level of class members

Declaration	Definition
public	Access not limited.
private	Access limited to the containing class.
internal	Access limited to this program.
protected	Access limited to the containing class and to types derived from the containing class
protected internal	Access limited to the containing class, derived classes, or to members of this program

Introduction

By using access modifiers, you can define the scope of class members in your applications. It is important to understand how access modifiers work because they affect your ability to use a class and its members.

Definition of scope

Scope refers to the region of code from which an element of the program can be referenced. For example, the **weight** member of the **Lion** class can be accessed only from within the **Lion** class. Therefore, the *scope* of the **weight** member is the **Lion** class.

Items that are nested within other items are within the scope of those items. For example, **Lion** is within the **ClassMain** class, and therefore can be referenced from anywhere within **ClassMain**.

C# access modifiers

The following table lists the access modifiers that can be added to your class members to control their scope at the time of declaration.

Declaration	Definition
public	Access is not limited: any other class can access a public member.
private	Access is limited to the containing type: only the class containing the member can access the member.
internal	Access is limited to this assembly: classes within the same assembly can access the member.
protected	Access is limited to the containing class and to types derived from the containing class.
protected internal	Access is limited to the containing class, derived classes, or to classes within the same assembly as the containing class.

An *assembly* is the collection of files that make up a program.

Rules

The following rules apply:

- Namespaces are always (implicitly) public.
- Classes are always (implicitly) public.
- Class members are private by default.
- Only one modifier can be declared on a class member. Although *protected internal* is two words, it is one access modifier.
- The scope of a member is never larger than that of its containing type.

Recommendations

The accessibility of your class members determines the set of behaviors that the user of your class sees. If you define a class member as private, the users of that class cannot see or use that member.

You should make public only those items that users of your class need to see. Limiting the set of actions that your class makes public reduces the complexity of your class from the point of view of the user, and it makes it easier for you to document and maintain your class.

Example 1

If a class named **Animal** contains a member named **weight**, the member is private by default and is accessible only from within the **Animal** class. If you try to use the **weight** member from another class, you get a compilation error, as shown in the following code:

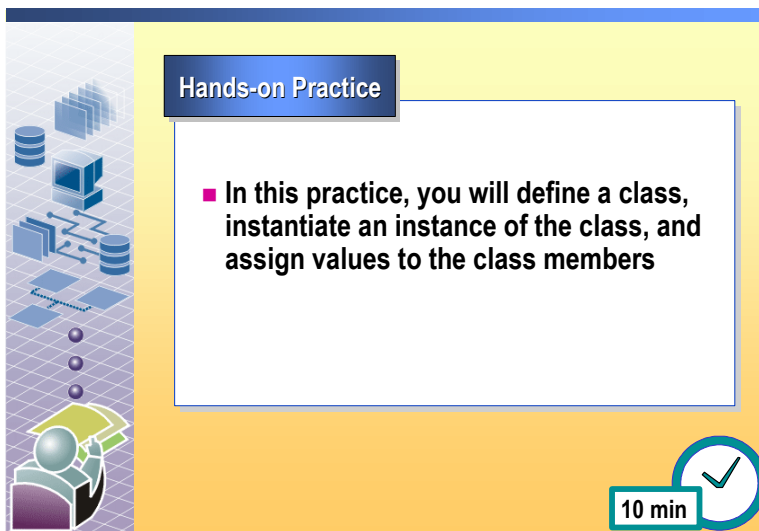
```
using System;
```

```
namespace LearnCSharp.ClassExample {  
    class ClassMain {  
  
        public class Lion {  
            public int    age;  
            private int    weight;  
        }  
  
        static void Main(string[] args) {  
            Lion zooLion = new Lion();  
            zooLion.age = 7;  
            // the following line causes a compilation error  
            zooLion.weight = 200;  
        }  
    }  
}
```

Compiling the preceding code produces the following error:

```
'LearnCSharp.ClassExample.ClassMain.Lion.weight' is  
inaccessible due to its protection level
```

Practice: Defining Classes and Creating Objects



Hands-on Practice

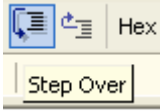
- In this practice, you will define a class, instantiate an instance of the class, and assign values to the class members

10 min

In this practice, you will create a class that represents an Antelope and create an instance of that class.

The solution code for this practice is located in *install_folder*\Practices\Mod03\Classes_Solution\ExampleClass.sln. Start a new instance of Microsoft Visual Studio® .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod03\Classes\ExampleClass.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod03\Classes, click ExampleClass.sln , and then click Open .
2. Review the tasks for this practice.	a. In Solution Explorer, click Form1.cs , and then press F7 to open the Code Editor. b. On the View menu, point to Show Tasks , and then click All . c. Review the tasks listed in the Task List.
3. Write code to define an Antelope class.	a. In the Task List, double-click TODO 1: Define an Antelope class . b. The Antelope class should have at least the following characteristics: <ul style="list-style-type: none"> Exhibit number Age

Tasks	Detailed steps
4. Create an instance of an Antelope object, and assign values to the members.	a. Locate the task TODO 2: Create an instance of the Antelope class . Any code that you place in the runExample_Click method will run when you click the Run button in the application window. b. Create an instance of the Antelope object, and assign a value to the Exhibit number member.
5. Use the supplied Output method to display the exhibit number for the Antelope object.	a. Use the provided Output method to display the information, replacing the word null in the provided example with the value that you want to display.
6. Compile your application, and then step through it in the debugging tool to ensure that your application is working as expected. 	a. Set a breakpoint at the line where you instantiate your first object. b. Press F5 to compile and run your application in debug mode. c. Click Run in your application. d. Step through your code by using the Step Over button, shown on the left, or by pressing F10.
7. Save your application, and then quit Visual Studio .NET.	a. Save your application. b. Quit Visual Studio .NET.

Lesson: Declaring Methods

- How to Write a Method
- How to Pass Parameters to a Method
- How to Pass Parameters by Reference
- How to Pass a Reference Type
- How to Overload a Method
- How to Use XML Code Comment Features

Introduction

This lesson explains how to implement actions in C# by using methods.

Lesson objectives

After completing this lesson, you will be able to:

- Write a method.
- Pass parameters to a method.
- Use the **ref** keyword to modify a parameter in a method.
- Use the **out** keyword to initialize a value in a method.
- Overload a method.
- Use the XML comment feature in Visual Studio .NET.

Lesson agenda

This lesson includes the following topics and activity:

- How to Write a Method
- How to Pass Parameters to a Method
- How to Pass Parameters by Reference
- How to Pass a Reference Type
- How to Overload a Method
- How to Use XML Code Comment Features
- Practice: Writing and Calling a Method

How to Write a Method

■ A method is a command for action

```
class Lion {
    private int weight;
    public bool IsNormalWeight () {
        if ((weight < 100)|| (weight > 250)) {
            return false;
        }
        return true;
    }
    public void Eat() { /* some action */ }
    public int GetWeight() {return this.weight;}
}

...
Lion bigLion = new Lion();
bool weightNormal = bigLion.IsNormalWeight();
bigLion.Eat();
int weight = bigLion.GetWeight();
```

Introduction

A method is a class member that is used to define the actions that can be performed by that object or class.

Syntax

The syntax for declaring a method is as follows:

[attributes] [modifiers] **return-type** **method-name** ([parameter-list]) statement-block

Rules

The following rules apply to methods:

- In the method declaration, you must always specify a return type. If the method is not designed to return a value to the caller, you specify a return type of **void**.
- Even if the method takes no arguments, you must include a set of empty parentheses after the method name.
- When calling a method, you must match the input parameters of the method exactly, including the return type, the number of parameters, their order, and their type. The method name and parameter list is known as the method *signature*.

Recommendation

The following are recommendations for naming methods:

- The name of a method should represent the action that you want to carry out. For this reason, methods usually have action-oriented names, such as **WriteLine** and **ChangeAddress**.
- Methods should be named using Pascal case.

Returning a value from a method

When you call a method, the execution jumps to that method and it executes until either a **return** statement or the end of the method is reached. At that point, the execution returns to the calling method.

When you want a method to return a value to the caller, use the **return** keyword followed by the value, where the type of the value is the same as the return-type of the method. If the return type of the method is **void**, you do not need to use the **return** keyword, or you can use return with no value specified.

Example

In the following code, the **IsNormalWeight** method checks the value of weight and returns **false** if the weight is outside the normal range. In the following example, **IsNormalWeight** must return **true** or **false**, because the **return-type** is **bool**:

```
class Lion {
    private int    weight;

    public bool IsNormalWeight() {
        if ( ( weight < 100 ) || ( weight > 250 ) ) {
            return false;
        }
        return true;
    }
    public void Eat() { }
    public int GetWeight() {
        return weight;
    }
}
```

You create the object as follows:

```
Lion bigLion = new Lion();
```

The **IsNormalWeight** method returns a Boolean value, so it can be used in an **if** statement as follows:

```
if ( bigLion.IsNormalWeight() == false ) {
    Console.WriteLine("Lion weight is abnormal");
}
```

The **Eat** method does not return a value, having a return type of **void**, so the following line of code simply instructs the **bigLion** object to perform the **Eat** action:

```
bigLion.Eat();
```

The **GetWeight** method returns an **int**, and it can be used to assign the resulting value to an int in the calling method as follows:

```
int bigLionWeight = bigLion.GetWeight();
```

The **this** keyword

The **this** keyword is used to refer to the current instance of an object. When **this** is used within a method, it allows you to refer to the members of the object.

For example, the **GetWeight** method can be modified to use **this** as follows:

```
public int GetWeight() {  
    return this.weight;  
}
```

In this case, the statement **this.weight** refers to the weight member of this object. This is functionally identical to the **GetWeight** method shown in the **Lion** class in the above example.

Using **this** can help to make your code more readable because it is immediately apparent to the person reading the code that you are referring to a member of this object. In addition, Microsoft IntelliSense® provides a list of class members when you type **this**.

Note The code used in this topic is available on the Student Materials compact disc in Samples\Mod03\Methods\Methods.sln.

How to Pass Parameters to a Method

■ Passing by value

```
class Lion {  
    private int weight;  
    public void SetWeight(int newWeight) {  
        weight = newWeight;  
    }  
}  
.  
.  
.  
  
Lion bigLion = new Lion();  
  
int bigLionWeight = 250;  
bigLion.SetWeight( bigLionWeight );
```

Introduction

When a value type variable is passed to a method, the method receives a copy of the value that was assigned to the variable. The method uses this value to perform an action.

Example 1

For example, given the following class:

```
class Lion {  
    private int weight;  
    public void SetWeight( int newWeight ) {  
        weight = newWeight;  
    }  
}
```

If you pass the value of **200** to the **SetWeight** method:

```
Lion bigLion = new Lion();
```

```
int bigLionWeight = 200;  
bigLion.SetWeight( bigLionWeight );
```

When the method is called, the value of **bigLionWeight** is copied to the *newWeight* parameter, and this changes the private member **weight** to **200**.

Example 2

In this example, two parameters are passed to an **Add** method that adds the numbers and returns the result. The result, 50, is assigned to the integer variable **total**, as shown in the following code:

```
class SimpleMath {  
    public int Add( int x, int y ) {  
        return x + y;  
    }  
}
```

```
SimpleMath sums = new SimpleMath();  
int total = sums.Add ( 20, 30 );
```

Passing by value

When you pass a variable as a parameter, the method works on a copy of that variable. This is called *passing by value*, because the value is provided to the method, yet the object that contains the value is not changed.

In the following example, the attempt to double the variable fails:

```
public void Double( int doubleTarget ) {  
    doubleTarget = doubleTarget * 2;  
}
```

```
int numbertoDouble = 10;  
sums.Double ( numbertoDouble);  
// numbertoDouble is still 10
```

When the **Double** method is called, the **numberToDouble** variable is copied into the *doubleTarget* parameter. This copy, in the **doubleTarget** variable, is a local variable within the scope of the **Double** method and is discarded when the method returns. The value in **numberToDouble** is unchanged.

The this keyword

An alternative implementation of the **SetWeight** method follows:

```
class Lion {  
    private int weight;  
    public void SetWeight( int weight ) {  
        this.weight = weight;  
    }  
}
```

In this example, the parameter *weight* has the same name as the class member. When *weight* is referenced within the scope of the method, the compiler will use the parameter value, so **this** is used to reference the class member.

This example illustrates the scope of parameters, and the use of the **this** keyword. You should make every attempt to avoid name collisions in your code.

Note The code used in this topic is available on the Student Materials compact disc in Samples\Mod03\ValueParameters\Value.sln.

How to Pass Parameters by Reference

■ Using the **ref** keyword

```
public void GetAddress(ref int number,
                      ref string street) {
    number = this.number;
    street = this.street;
}

. . .
int sNumber = 0; string streetName = null;
zoo.GetAddress( ref sNumber, ref streetName );
// sNumber and streetName have new values
```

■ Definite assignment

■ Using the **out** parameter keyword

- Allows you to initialize a variable in a method

Introduction

Methods return only a single value, but sometimes you want a method to modify or return multiple values. You can achieve this by passing the method a *reference* to the variable that you want to modify. The method can use the reference to access the actual value and change the value.

When a value is passed by reference, the method receives a reference to the actual value—so any changes that the method makes to the variable are actually made to the object that was passed to the method.

The **ref** keyword

You declare that a parameter is a reference parameter by using the **ref** keyword. Use the **ref** keyword in the parameter list to indicate to the compiler that the value is being passed by reference. You also must use the **ref** keyword when you call the method.

Non-example

Suppose that you have a class representing a Zoo that attempts to implement a **GetAddress** method, as shown in the following code:

```
class Zoo {
    private int    streetNumber = 123;
    private string streetName = "High Street";
    private string cityName = "Sammamish";
    public void GetAddress(int number, string street,
                           string city) {

        number = streetNumber;
        street = streetName;
        city = cityName;
    }
}
```

The attempt to retrieve the address fails, as shown in the following code:

```
Zoo localZoo = new Zoo();
int zooStreetNumber = 0;
string zooStreetName = null;
string zooCity = null;
localZoo.GetAddress(zooStreetNumber, zooStreetName, zooCity);
// zooStreetNumber, zooStreetName, and zooCity still 0 or null
```

The **GetAddress** method in the preceding example operates on copies of the parameters, so **zooStreetNumber** is still **0** and **zooStreetName** and **zooCity** are still null after the method is called.

Example

Using the **ref** keyword allows this code to work as planned:

```
public void GetAddress( ref int number,
                      ref string street,
                      ref string city) {
    number = streetNumber;
    street = streetName;
    city = cityName;
}
```

You must also use the **ref** keyword when you call the function:

```
localZoo.GetAddress(ref zooStreetNumber, ref zooStreetName,
ref zooCity);

if ( zooCity == "Sammamish" ) {
    Console.WriteLine("City name was changed");
}
```

The variables **zooStreetNumber**, **zooStreetName**, and **zooCity** are assigned the values that are stored in the object.

Definite assignment

C# imposes definite assignment, which requires that all variables are initialized before they are passed to a method. This eliminates a common bug caused by the use of unassigned variables.

Even if you have a variable that you know will be initialized within a method, definite assignment requires that you initialize the variable before it can be passed to a method. For example, in the preceding code, the lines that initialize **zooStreetNumber**, **zooStreetName**, and **zooCity** are required to make the code compile, even though the intent is to initialize them in the method.

The out keyword

By using the **out** keyword, you can eliminate the redundant initialization. Use the **out** keyword in situations where you want to inform the compiler that variable initialization is occurring within a method. When you use the **out** keyword with a variable that is being passed to a method, you can pass an uninitialized variable to that method.

Example

The following code modifies the zoo **GetAddress** method to demonstrate use of the **out** keyword:

```
using System;

namespace LearnCSharp.MethodExample1 {
    class Zoo {
        private int    streetNumber = 123;
        private string streetName = "High Street";
        private string cityName = "Sammamish";

        public void GetAddress(    out int number,
                                out string street,
                                out string city) {
            number = streetNumber;
            street = streetName;
            city = cityName;
        }
    }

    class ClassMain {
        static void Main(string[] args) {
            Zoo localZoo = new Zoo();
            // note these variables are not initialized
            int    zooStreetNumber;
            string zooStreetName;
            string zooCity;
            localZoo.GetAddress(    out zooStreetNumber,
                                out zooStreetName,
                                out zooCity);

            Console.WriteLine(zooCity);
            // Writes "Sammamish" to a console
        }
    }
}
```

How to Pass a Reference Type

- When you pass a reference type to a method, the method can alter the actual object

```
class Zoo {  
    public void AddLion( Lion newLion ) {  
        newLion.location = "Exhibit 3";  
        . . .  
    }  
}  
  
. . .  
  
Zoo myZoo = new Zoo();  
Lion babyLion = new Lion();  
myZoo.AddLion( babyLion );  
// babyLion.location is "Exhibit 3"
```

Introduction

When you pass a reference type variable to a method, the method can alter the actual value because it is operating on a reference to the *same object*.

Example 1

In the following example, a **babyLion** object is passed to an **AddLion** method, where the **location** member of **babyLion** is assigned the value **Exhibit 3**. Because the reference to the actual object is passed to the method, the method can change the value of **location** in the **babyLion** object.

using System;

```
namespace LearningCSharp {  
    class MainClass {  
        static void Main(string[] args) {  
            Zoo myZoo = new Zoo();  
            Lion babyLion = new Lion();  
  
            myZoo.AddLion( babyLion );  
            //babyLion.location is Exhibit 3  
        }  
    }  
  
    class Lion {  
        public string location;  
    }  
  
    class Zoo {  
        public void AddLion( Lion newLion ) {  
            newLion.location = "Exhibit 3";  
        }  
    }  
}
```

Example 2

The following code defines an **Address** class. It creates an **Address** object, **zooLocation**, in the **Main** method and passes the object to the **GetAddress** method. Because **Address** is a reference type, **GetAddress** receives a reference to the same object that was created in **Main**. **GetAddress** assigns the values to the members, and when control returns to **Main**, the **zooLocation** object has the address information.

```
using System;

namespace LearningCSharp {
    class Address {
        public int number;
        public string street;
        public string city;
    }

    class Zoo {
        private int    streetNumber = 123;
        private string streetName = "High Street";
        private string cityName = "Sammamish";

        public void GetAddress( Address zooAddress ) {
            zooAddress.number = streetNumber;
            zooAddress.street = streetName;
            zooAddress.city = cityName;
        }
    }

    class ClassMain {
        static void Main(string[] args) {
            Zoo localZoo = new Zoo();
            Address zooLocation = new Address();

            localZoo.GetAddress( zooLocation );

            Console.WriteLine( zooLocation.city );
            // Writes "Sammamish" to a console
        }
    }
}
```

Example 3

You can use different types of parameters in a single method. For example, you may want to request a specific exhibit for the new lion.

```
public void AddLion(Lion newLion, int preferredExhibit) { }
```

How to Overload a Method

- **Overloading enables you to create multiple methods within a class that have the same name but different signatures**

```
class Zoo {  
    public void AddLion(Lion newLion) {  
        ...  
    }  
    public void AddLion(Lion newLion,  
                        int exhibitNumber) {  
        ...  
    }  
}
```

Introduction

When calling a method, you must match the input parameters exactly; including the return type, the number of parameters, and their order.

Definition

Method overloading is a language feature that enables you to create multiple methods in one class that have the same name but that take different signatures.

By overloading a method, you provide the users of your class with a consistent name for an action while also providing them with several ways to apply that action.

Example 1

Your **Zoo** class includes an **AddLion** method that allows you to add new **Lion** objects, but sometimes you want to place them in a specific exhibit and other times you do not want to specify this information. You can write two **AddLion** methods, one that accepts a *Lion* parameter, and one that accepts *Lion* and exhibit number parameters.

```
class Zoo {  
    public void AddLion( Lion newLion ) {  
        // Place lion in an appropriate exhibit  
    }  
    public void AddLion( Lion newLion, int exhibitNumber ) {  
        // Place the lion in exhibitNumber exhibit  
    }  
}
```

When you subsequently call the **AddLion** method, you call the correct method by matching the parameters, as shown in the following code:

```
Zoo myZoo = new Zoo();  
Lion babyLion = new Lion();  
  
myZoo.AddLion( babyLion );  
myZoo.AddLion( babyLion, 2 );
```


Example 2

Suppose you have a Web site that allows people to sign up for a newsletter about a zoo. Some information is necessary, such as name and e-mail address, but some is optional, such as favorite animal. You can use one method name to handle this situation, as shown in the following code:

```
class ZooCustomer {
    private string name;
    private string email;
    private string favoriteAnimal;

    public void SetInfo( string webName,
                        string webEmail,
                        string animal ) {
        name = webName;
        email = webEmail;
        favoriteAnimal = animal;
    }

    public void SetInfo(string webName, string webEmail) {
        name = webName;
        email = webEmail;
    }
}
```

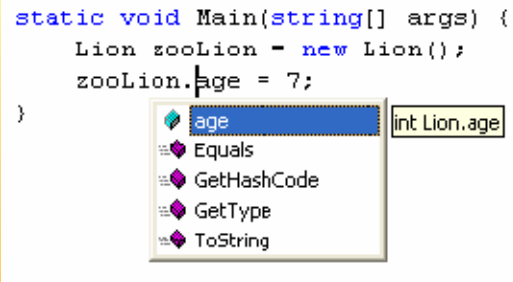
Why overload?

Consider the following guidelines as you decide whether to use method overloading:

- Use overloading when you have similar methods that require different parameters.
- Overloaded methods are a good way for you to add new functionality to existing code.
- Use overloaded methods only to implement methods that provide similar functionality.

How to Use XML Code Comment Features

- Three forward slashes (///) inserts XML comments
- Hovering over a section of code produces a pop-up menu for accessing class member information

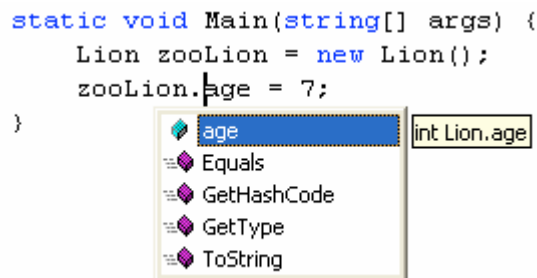


Introduction

Visual Studio .NET provides several useful features for working with classes and accessing class members. These features include pop-up menus for accessing class member information, and an XML code comment feature.

Pop-up menus

For example, in the Code Editor, when you type the dot operator after an object name, Visual Studio .NET displays all the members of that class in a list from which you can select the member you want to access, as shown in the following illustration:



You can also access this list from the **Edit** menu by pointing to **IntelliSense** and then clicking **List Members**, or by pressing CTRL+J.

XML comments

Visual Studio .NET also provides an XML code comment feature that makes it easy for you to include useful comments in your code.

When you type three forward slashes (///), Visual Studio.NET inserts several lines of XML code for you, and all you have to do is enter the actual description of the type and type members. Even the correct values for the parameter names are included for you in the `<param>` tags.

The following table shows the <param> tags that you can use:

Declaration	Definition
<summary>	One line summary of a class, method, or property
<remarks>	May contain <para> and <list> formatting tags, as well as <cref> for creating hyperlinks
<value>	Description of a property
<exception>	Exceptions arising from methods and properties
<param>	Method parameters

IntelliSense enables you to hover your mouse pointer over a section of code to reveal the comments and type information.

Adding comments to a method

Use the following procedures to include XML comments in your code:

1. Type /// on the line above the method.

```
///
public void SetInfo(string webName, string webEmail) {
    name = webName;
    email = webEmail;
}
```

2. Examine the XML template that is provided by Visual Studio .NET.
3. Document your method.

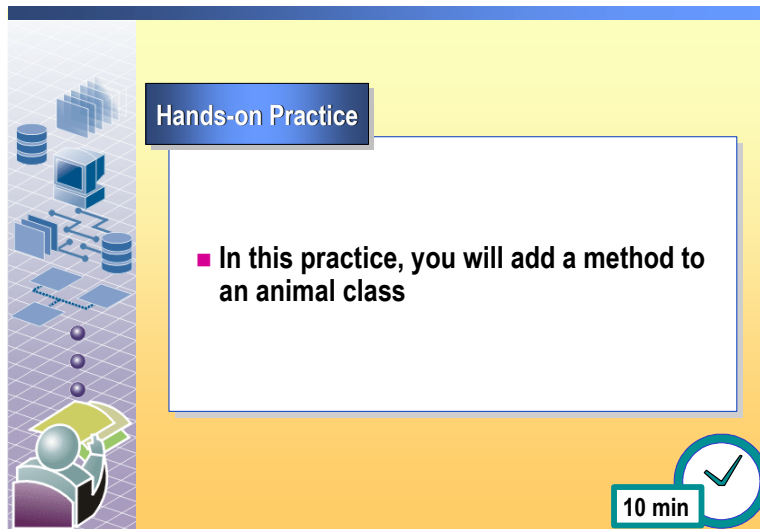
```
/// <summary>
/// Sets customer information from the Web page
/// </summary>
/// <param name="webName">Name supplied by customer</param>
/// <param name="webEmail">Email supplied by customer</param>
public void SetInfo(string webName, string webEmail) {
    name = webName;
    email = webEmail;
}
```

4. Use the method to see how your code comments are integrated with IntelliSense.

```
SetInfo (
void Form1.SetInfo (string webName, string webEmail)
webName:
    Name supplied by customer
```

The C# compiler can extract the XML elements from your comments and generate an XML file for you.

Practice: Writing and Calling a Method



Hands-on Practice

- In this practice, you will add a method to an animal class

10 min

In this practice, you will add a method to an animal class.

The solution code for this practice is provided in *install_folder*\Practices\Mod03\Methods\MethodExample.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod03\Methods\MethodExample.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod03\Methods, click MethodExample.sln , and then click Open .
2. Review the tasks for this practice.	a. In Solution Explorer, click Form1.cs , and then press F7 to open the Code Editor. b. On the View menu, point to Show Tasks , and then click Comment .
3. Write a method in the Antelope class that sets the private member weight to a value that is specified in a parameter.	a. In the Task List, double-click TODO 1: Write a SetWeight method, that sets the weight member . b. In the Antelope class, write a SetWeight method that takes one parameter, representing the weight of the animal. c. Use the parameter that is passed to the method to set the private member weight in the Antelope class.

Tasks	Detailed steps
4. Call the SetWeight method with a weight of 100 .	a. Locate the task TODO 2: Call the method to set the Antelope's weight . b. In the runExample_Click method, write code that calls the SetWeight method of the babyAntelope object, with the value 100 as a parameter.
5. Compile your program and run it. The value 500 should be displayed in the text window.	a. Press F5 to compile and run your application. b. In your application window, click Run . c. Ensure that the value 100 appears in the text window.
6. Save your application, and then quit Visual Studio .NET.	a. Save your application. b. Quit Visual Studio .NET.

Optional:

Add a **GetWeight** method to the **Antelope** class. **GetWeight** should take no parameters and return the weight of the **babyAntelope** object. If you do this, make the **weight** member private.

Modify the **Output** method so that it uses the **GetWeight** method.

Lesson: Using Constructors

- How to Initialize an Object
- How to Overload a Constructor

Introduction

This lesson defines class constructors and explains how to use them to initialize objects.

Lesson objectives

After completing this lesson, you will be able to:

- Write constructors.
- Overload constructors.

Lesson agenda

This lesson includes the following topics and activity:

- How to Initialize an Object
- How to Overload a Constructor
- Practice: Using Constructors

How to Initialize an Object

- **Instance constructors are special methods that implement the actions required to initialize an object**

- Have the same name as the name of the class
- Default constructor takes no parameters

```
public class Lion {  
    public Lion() {  
        Console.WriteLine("Constructing Lion");  
    }  
}
```

- **Readonly**

- Used to assign a value to a variable in the constructor

Introduction

Every class implicitly or explicitly includes an *instance constructor*, which is a method that is automatically called by the runtime whenever an instance of the class is created.

Definition

Constructors are special methods that implement the actions that are required to initialize an object.

Creating a class constructor

The constructor method is defined by using the same name as the class in which it is declared.

Syntax

[attributes] [modifiers] **constructor-name** ([parameters])
[initializer] statement-block

For example, the following code contains a constructor:

```
public class Lion {  
    public Lion() {  
        Console.WriteLine("Constructing Lion");  
    }  
}
```

When the following code is executed, the Lion constructor is called when the object is instantiated:

```
Lion babyLion = new Lion();  
Console.WriteLine("Made a new Lion object");
```

The following output is produced:

```
Constructing Lion  
Made a new Lion object
```

If you do not write an instance constructor, C# automatically provides a default instance constructor. For example, you can write a class as follows:

```
public class Lion {  
    private string name;  
}
```

This is exactly equivalent to:

```
public class Lion {  
    private string name;  
    public Lion() {  
    }  
}
```

Constructor parameters

Constructors can take parameters, like any other method. When you specify a constructor with a parameter, the default constructor for that object is not provided. For example, if you want users of your class to always specify the name of the lion when they create it, you can write a constructor as shown in the following code:

```
public class Lion {  
    private string name;  
    public Lion( string newLionName ) {  
        this.name = newLionName;  
    }  
}
```

When users create this object, they must specify a name:

```
Lion babyLion = new Lion("Leo");
```

Failure to provide a name will result in a compilation error.

Class initialization

When an object is created, the instance members in the class are implicitly initialized. For example, in the following code, **zooName** is initialized to **null**:

```
class Zoo {  
    public string zooName;  
}
```

Usually, the purpose of writing your own constructor is to perform some initialization of the members of the object. For example:

```
class Zoo {  
    public string zooName;  
    public Zoo() {  
        zooName = "Sammamish Zoo";  
    }  
}
```

The **zooName** member of every instance of the **Zoo** class is now set to "Sammamish Zoo". When a new **Zoo** class is instantiated:

```
Zoo localZoo = new Zoo();  
Console.WriteLine(localZoo.zooName);
```


The following output is produced:

```
Sammamish Zoo
```

A better way to write this class is as follows:

```
class Zoo {
    public string zooName = "Sammamish Zoo";
}
```

When an instance of **Zoo** is created, the instance member variables are initialized. Because any values are assigned before the constructor is executed, the constructor can use the values.

```
class Zoo {
    public string zooName = "Sammamish Zoo";
    public Zoo() {
        if ( zooName == "Duwamish Zoo" ) {
            // This can never happen
        }
    }
}
```

readonly

When you use the **readonly** modifier on a member variable, you can only assign it a value when the class or object initializes, either by directly assigning the member variable a value, or by assigning it in the constructor.

Use the **readonly** modifier when a **const** keyword is not appropriate because you are not using a literal value—meaning that the actual value of the variable is not known at the time of compilation.

In the following example, the value of **admissionPrice** is set in the constructor, and subsequent attempts to set the value will fail, resulting in a compilation error.

```
class Zoo {
    private int            numberAnimals;
    public readonly decimal admissionPrice;

    public Zoo() {
        // Get the numberAnimals from some source...
        if ( numberAnimals > 50 ) {
            admissionPrice = 25;
        }
        else {
            admissionPrice = 20;
        }
    }
}
```

How to Overload a Constructor

- Create multiple constructors that have the same name but different signatures

- Specify an initializer with **this**

```
public class Lion {  
    private string    name;  
    private int       age;  
  
    public Lion() : this( "unknown", 0 ) {  
        Console.WriteLine("Default: {0}", name);  
    }  
    public Lion( string theName, int theAge ) {  
        name = theName;  
        age = theAge;  
        Console.WriteLine("Specified: {0}", name);  
    }  
}
```

Introduction

It is often useful to overload a constructor to allow instances to be created in more than one way.

Syntax

You overload a constructor in the same way that you overload a method: create a base class that contains two or more constructors with the same name but different input parameters.

Example 1

For example, if you create a new record for an adopted lion, you can create different constructors depending upon the information that is available at the time that you create the record, as shown in the following code:

```
class Lion {
    private string name;
    private int age;

    public Lion( string theName, int theAge ) {
        name = theName;
        age = theAge;
    }

    public Lion( string theName ) {
        name = theName;
    }

    public Lion( int theAge ) {
        age = theAge;
    }
}

Lion adoptedLion = new Lion( "Leo", 3 );
Lion otherAdoptedLion = new Lion( "Fang" );
Lion newbornLion = new Lion( 0 );
```

Now, when you create a lion record, you can add the information that you have available. You can assume that other methods exist to allow you to update the object information.

Specifying an initializer

Often when you have multiple constructors, you initialize each one in a similar manner. Rather than repeating the same code in each constructor, attempt to centralize common code in one constructor and call that from the other constructors.

To call a specific constructor that is defined in the class itself, use the **this** keyword. When you add **this** to the constructor declaration, the constructor that matches the specified parameter list (has the same signature) is invoked. An empty parameter list invokes the default constructor.

Example 2

```
public class Lion {
    private string name;
    private int    age;

    public Lion() : this ( "unknown", 0 ) {
        Console.WriteLine("Default {0}", name);
    }

    public Lion( string theName, int theAge ) {
        name = theName;
        age = theAge;
        Console.WriteLine("Specified: {0}", name);
    }
}

. . .

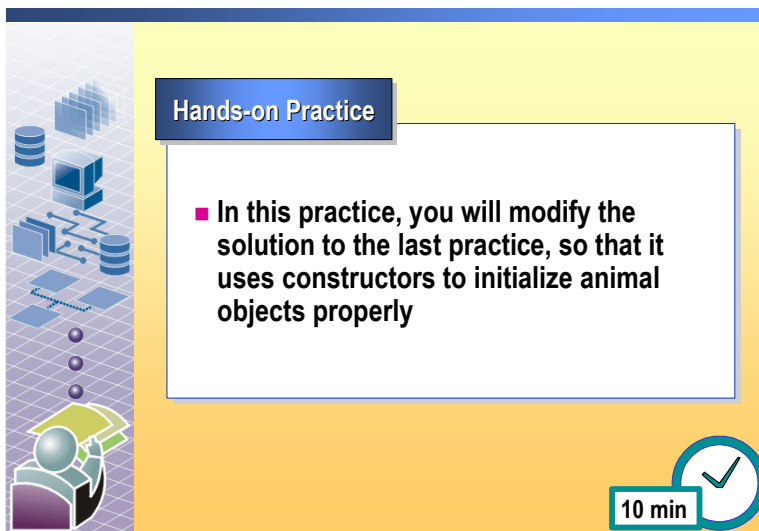
Lion adoptedLion = new Lion();
```

In this example, when the user of the class creates the **adoptedLion** object, the class invokes the matching constructor, which is **Lion()**. Before **Lion()** executes the code in the body of the constructor, it invokes an alternative instance constructor that has parameters matching those specified in the **this** initializer (a **string** and an **int**). Before any of the code in the constructors is executed, however, the member variables are initialized and assigned.

The output from this sample is:

```
Specified unknown
Default unknown
```

Practice: Using Constructors



Hands-on Practice

■ In this practice, you will modify the solution to the last practice, so that it uses constructors to initialize animal objects properly

10 min

In this practice, you will modify the solution to the last practice, so that it uses constructors to initialize animal objects properly.

The solution code for this practice is provided in *install_folder*\Practices\Mod03\Ctor_Solutions\CtorExample.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod03\Ctor\CtorExample.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod03\Ctor, click CtorExample.sln , and then click Open .
2. Review the tasks for this practice.	a. In Solution Explorer, click Form1.cs , and then press F7 to open the Code Editor. b. On the View menu, point to Show Tasks , and then click Comment .
3. Specify the antelope's gender in the constructor call.	<ul style="list-style-type: none"> ■ In the Task List, double-click TODO 1: Change the constructor call to specify the gender. <ul style="list-style-type: none"> • Specify the gender as a string, for example “male”.

Tasks	Detailed steps
4. Test your code and observe the error.	<ul style="list-style-type: none"> a. Press F5 to compile and run your application. b. You will receive an error. This is because the Antelope class does not contain a constructor that takes a parameter.
5. Write an Antelope constructor that accepts a string.	<ul style="list-style-type: none"> a. In the Task List, double-click TODO 2: Add a constructor. b. Add a constructor that accepts the string that you added in step 3, and uses it to set the Antelope member gender.
6. Test your code and verify that a male antelope is created.	<ul style="list-style-type: none"> a. Press F5 to compile and run your application. b. In the application window, click Run, and verify that you receive a message stating that a male antelope object was created.
7. Save your application, and then quit Visual Studio .NET.	<ul style="list-style-type: none"> a. Save your application. b. Quit Visual Studio .NET.

Lesson: Using Static Class Members

- How to Use Static Class Members
- How to Initialize a Class

Introduction

This lesson introduces you to static class members. Static members belong to the class, rather than an instance. Static constructors are used to initialize a class.

Lesson objective(s)

After completing this lesson, you will be able to:

- Use static class members.
- Initialize a class using a static constructor.

Lesson Agenda

This lesson includes the following topics and activity:

- How to Use Static Class Members
- How to Initialize a Class
- Practice: Using Static Class Members

How to Use Static Class Members

■ Static Members

- Belong to the class
- Initialize before an instance of the class is created
- Shared by all instances of the class

```
class Lion {  
    public static string family = "felidae";  
}  
...  
// A Lion object is not created in this code  
Console.WriteLine( "Family: {0}", Lion.family );
```

Introduction

Classes can have static members, such as properties, methods and variables.

Static members are associated with the class, not with a specific instance of the class. Static members are useful when you want to initialize or provide a value that is shared by all instances of a class.

Static members

Because static members belong to the class, rather than an instance, they are accessed through the class, not through an instance of the class. The following complete code example shows how to use the static member **family**.

```
using System;
```

```
namespace StaticExample {  
    class ZooDemo {  
        static void Main(string[] args) {  
            Console.WriteLine( "Family: {0}", Lion.family );  
            Console.ReadLine();  
        }  
    }  
  
    class Lion {  
        public static string family = "felidae";  
    }  
}
```

This code samples produces the following output:

felidae

Static methods

Methods can also be static. When the static access modifier is applied to a method, the method is accessible only through the class, not an object instance.

Because static methods are part of the class, you can invoke them without creating an instance of the object. In C#, you cannot access a static method from an instance.

The static modifier provides global methods. When you add a static declaration to a method, you declare that there will be only one copy of the method, no matter how many times that class is created.

Use static members when they refer to or operate on information that is about the class, rather than about an instance of a class. For example, you can use a static method to maintain a count of the number of objects that are created from a class, or to log information about objects of a specific class.

The following example counts the number of male or female lions that are added to the Zoo:

```
using System;

namespace StaticExample {
    enum Gender {
        Male,
        Female
    }

    class ZooDemo {
        static void Main(string[] args) {
            Lion male1 = new Lion( Gender.Male );
            Lion male2 = new Lion( Gender.Male );
            Lion male3 = new Lion( Gender.Male );

            Console.WriteLine("Males {0}", Lion.NumberMales() );
        }
    }

    class Lion {
        private static int males;
        private static int females;

        public Lion(Gender lionGender) {
            if ( lionGender == Gender.Male ) {
                males++;
            }
            else {
                females++;
            }
        }

        public static int NumberMales() {
            return males;
        }
    }
}
```

The preceding code is provided on the Student Materials compact disc in the Samples\Mod03\Static folder.

How to Initialize a Class

■ Static Constructors

- Will only ever be executed once
- Run before the first object of that type is created
- Have no parameters
- Do not take an access modifier
- May co-exist with a class constructor
- Used to initialize a class

Introduction

Instance constructors are used to initialize an object. You can, however, write a constructor that initializes a class. This type of constructor is called a *static constructor*. You create a static constructor by using a **static** modifier.

Static constructor

A static constructor is sometimes referred to as a *shared* or *global* constructor because it does not operate on a specific instance of a class.

You cannot call a static constructor directly. It is executed at most *once* before the first instance of the class is created or before any static methods are used. Therefore, a static constructor is useful for initializing values that will be used by all instances of the class.

Syntax

Like instance constructors, static constructors have the same name as the class, and an empty parameter list. You declare a static constructor by using the **static** modifier. It does not take an access modifier and it can coexist with an instance constructor.

```
class Lion {  
    static Lion() {  
        // class-specific initialization  
    }  
}
```

Example 1

For example, suppose that there are several zoo animal classes, each of which has a class member **family**. All Lions belong to the family *felidae*, so it is useful to set this information for the class.

```
class Lion {
    static private string family;
    static Lion() {
        family = "felidae";
    }
}
```

Example 2

For example, if your code must initialize a series of values that are required for a calculation, or load a set of data that will be used by all instances of the class, such as a look-up table, then it can be useful to perform this task only once for the class, rather than every time an instance is created.

The following example uses **System.Random**, the pseudo-random number generator that is provided in the .NET Framework, to generate random numbers. It creates a single instance of **Random**, and every instance of the **RandomNumberGenerator** class uses this object.

```
using System;

namespace StaticConstructor {

    class RandomNumberGenerator {
        private static Random randomNumber;

        static RandomNumberGenerator() {
            randomNumber = new Random();
        }

        public int Next() {
            return randomNumber.Next();
        }
    }

    class Class1 {
        static void Main(string[] args) {
            RandomNumberGenerator r
                = new RandomNumberGenerator();

            for ( int i = 0; i < 10; i++ ) {
                Console.WriteLine( r.Next() );
            }
        }
    }
}
```

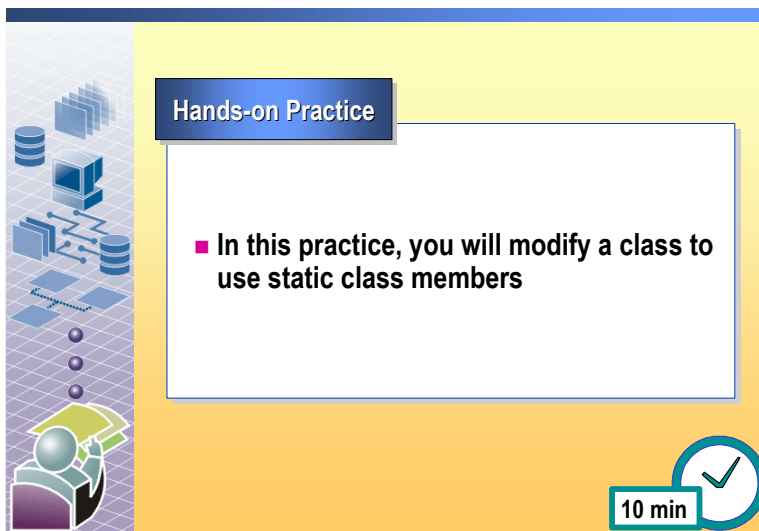
In this example, the **randomNumber** member of the class is declared as **static**. This member variable must be static for the static constructor to be able to assign a value to it. Instance members are not initialized until an instance of the class is created, so an attempt to assign to an instance variable from a static constructor results in a compilation error.

For example, the following code will not compile, because **firstNumber** does not exist when the constructor is called.

```
class RandomNumberGenerator {
    private static Random randomNumber;
    private int          firstNumber; // ERROR!

    static RandomNumberGenerator() {
        randomNumber = new Random();
        firstNumber = randomNumber.Next();
    }
}
```

Practice: Using Static Class Members



In this practice, you will maintain a count of the number of **Antelope** objects that are created, by adding a static member to the **Antelope** class and incrementing it in the antelope constructor.

The solution code for this practice is located in *install_folder*\Practices\Mod03\Static_Solution\StaticExample.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod03\Static\StaticExample.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod03\Static, click StaticExample.sln , and then click Open .
2. Review the tasks for this practice.	a. In Solution Explorer, click Form1.cs , and then press F7 to open the Code Editor. b. On the View menu, point to Show Tasks , and then click Comment .
3. Add an public static int variable to the Antelope class, and name it numberOfAntelopes .	a. In the Task List, double-click TODO 1: Add a public static int called numberOfAntelopes . b. Add the following line of code to the class: <pre>public static int numberOfAntelopes;</pre>

Tasks	Detailed steps
4. In the Antelope constructor, increment the numberOfAntelopes member variable.	a. Locate the task TODO 2: Increment the numberOfAntelopes variable . b. Every time an Antelope object is created, increment the numberOfAntelopes variable by adding the following code to the constructor: <code>numberOfAntelopes++;</code>
5. Display the number of antelopes that are created.	a. Locate the task TODO 3: Display the number of antelopes created . b. Replace the null parameter in the Output method with a reference to the static numberOfAntelopes method. For example: <code>Output("Number of Antelopes: " + Antelope.numberOfAntelopes);</code>
6. Test your code.	a. Set a breakpoint at the first statement in the runExample_Click method. b. Press F5 to compile and run your application. c. In your application window, click Run . d. Step through the code and ensure that your program is functioning as expected. <p>Note that the numberOfAntelopes variable is shared by both of the Antelope objects.</p> e. When you are finished, stop debugging by closing your application or pressing SHIFT+F5.
7. Save your application, and quit Visual Studio .NET.	a. Save your application. b. Quit Visual Studio .NET.

Review

- Defining a Class
- Declaring Methods
- Using Constructors
- Using Static Class Members

1. What is the default accessibility level of a class member?
 - a. Public
 - b. Private
 - c. Internal

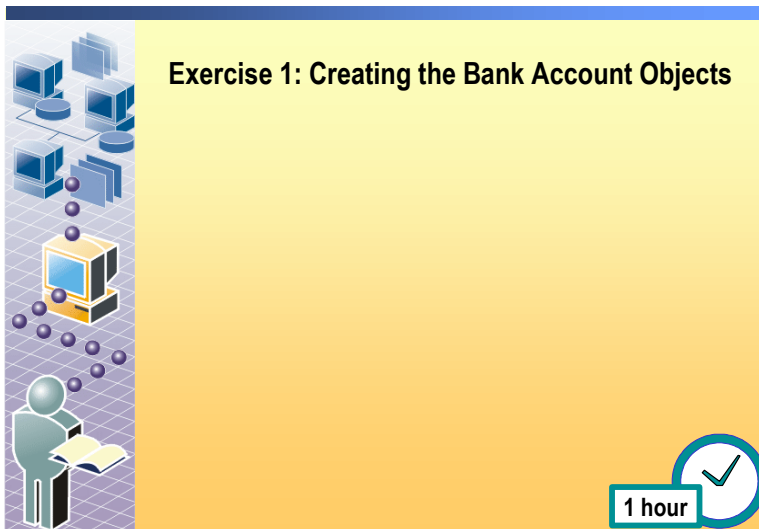
2. What is the keyword that is used to inform the compiler that a variable is initialized within a method?

3. What is the purpose of overloading a constructor?

4. When and how often does a static constructor execute?

5. Can you invoke a static method without instantiating an object? Why or why not?

Lab 3.1: Creating Classes in C#



Objectives

After completing this lab, you will be able to:

- Create classes and objects.
- Write methods.
- Pass parameters to methods.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Scenario

You are a programmer at a bank and have been asked to define objects for the types of bank accounts that customers can open. These accounts are:

- Checking account
- Savings account

Checking account

Each checking account has the following characteristics:

- The account holder's name can be assigned only when the account is created.
- The opening balance must be specified when the account is created.
- The account number must be assigned when the account is created.

Checking account numbers range from 100000 to 499999, and every checking account must have a unique account number. You do not need to check the upper limit of the account number in this lab.

A checking account holder can:

- Order a checkbook.
- Check the account balance.
- Add money to the checking account.
- Withdraw money if the account has sufficient funds.

Savings account

A savings account has the following characteristics:

- The account holder's name can be assigned only when the account is created.
- Saving account numbers range from 500000 to 999999. You do not need to check the upper limit of the account number in this lab.
- The account earns interest.

The interest rate depends on the account balance. If the balance is above 1000, the rate is 6%; otherwise, it is 3%.

A savings account holder can:

- Check the account balance.
- Add money to the account.
- Withdraw money if the account has sufficient balance.

**Estimated time to
complete this lab:
60 minutes**

Exercise 0

Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
<ul style="list-style-type: none">Log on to Windows as Student with a password of P@ssw0rd.	<ul style="list-style-type: none">Log on to Windows with the following account:<ul style="list-style-type: none">User name: StudentPassword: P@ssw0rd <p>Note that the 0 in the password is a zero.</p>

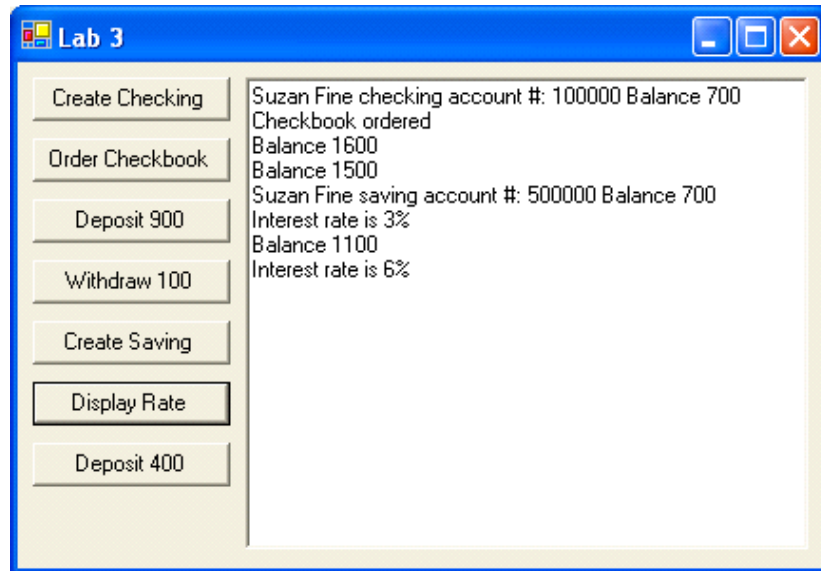
Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

Exercise 1

Creating the Bank Account Objects

In this exercise, you will write the objects that represent the bank account classes that are outlined in the scenario.

A sample solution is shown in the following illustration:



The solution code for this lab is located at *install_folder*\Labfiles\Lab03_1\Exercise1\Solution_Code. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Microsoft Visual Studio .NET, and then open <i>install_folder</i> \Labfiles\Lab03_1\Exercise1\Bank.sln.	<ul style="list-style-type: none">a. Start a new instance of Visual Studio .NET.b. On the Start Page, click Open Project.c. In the Open Project dialog box, browse to <i>install_folder</i>\Labfiles\Lab03_1\Exercise1, click Bank.sln, and then click Open. Form1.cs provides the user interface.
2. In Solution Explorer, locate the BankAccount.cs file.	<ul style="list-style-type: none">■ In Solution Explorer, click the C# file BankAccount.cs, and then press F7 to open the Code Editor. <p>This file is provided as a place for you to implement the bank account class or classes.</p>

Tasks	Detailed steps
3. Write the checking account class.	<p>a. Define a class named CheckingAccount.</p> <p>b. Write a constructor that implements the following rules:</p> <ul style="list-style-type: none"> • An account holder's name can be assigned only when the account is created. • The opening balance must be specified when the account is created. • An account number must be assigned when the account is created. Checking account numbers range from 100000 to 499999, and every checking account must have a unique account number. You do not need to check the upper limit of the account number in this lab. Use a static member to implement this rule.
4. Test your code by performing the actions listed on the right. Place your test code in the methods that are provided in the Form1 class in the sample code.	<p>a. In the checking_Click method in the Form1 class, create a new checking account for the customer Suzan Fine, with a balance of 700. The object should be declared in Form1, but not in the checking_Click method. Look for the comment //TODO: place bank account objects here.</p> <p>b. Display the information about the account in the text box window by using the provided Output method.</p> <p>c. Run your application by pressing F5, and then in the application window, click Create Checking.</p>
5. In the CheckingAccount class, write a method to order a checkbook.	<p>a. In the CheckingAccount class, write a method named OrderCheckBook that always returns true.</p> <p>b. Call the OrderCheckBook method from the Form1 class, in the checkbook_Click method.</p> <p>c. Compile and test your application.</p>
6. Write Deposit and Withdraw methods in the CheckingAccount class.	<p>a. In the CheckingAccount class, write a method named Deposit that adds money to the account.</p> <p>b. In the CheckingAccount class, write a method named Withdraw that removes money from the account. Do not permit the balance to fall below zero, and return false if the withdrawal amount is greater than the balance.</p>
7. Call the Withdraw and Deposit methods from the Form1 class.	<p>a. In the deposit_Click method in the Form1 class, write code to deposit 700 into the checking account, and display the new balance.</p> <p>b. In the withdraw_Click method in the Form1 class, write code to withdraw 100 from the checking account, and display the new balance.</p>

Tasks	Detailed steps
8. Write a SavingAccount class.	<ul style="list-style-type: none"> ▪ In the BankAccount.cs file, define a SavingAccount class that implements the following rules: <ul style="list-style-type: none"> • An account holder's name can be assigned only when the account is created. Savings account numbers range from 500000 to 999999. You do not need to check the upper limit of the account number in this lab. Hint: use a static member to manage the account numbers. • The interest rate depends on the account balance. If the balance is above 1000, the rate is 6%; otherwise it is 3%.
9. Create a SavingAccount object.	<ul style="list-style-type: none"> a. In the saving_Click method in the Form1 class, create a new checking account for the customer Suzan Fine, with a balance of 700. The object should be declared in Form1, but not in the saving_Click method. Look for the comment //TODO: place bank account objects here. b. Display the information about the account in the text box window by using the provided Output method. c. Run your application by pressing F5, and then in the application window click Create Saving.
10. Write a GetRate method for the SavingAccount class, that returns the current interest rate, and call it from the interest_Click method in Form1 .	<ul style="list-style-type: none"> a. In the SavingAccount class, write a method that returns the current interest rate. b. In the interest_Click method in Form1, call the SavingAccount method, and display the current interest rate by using the Output method. c. Compile, run, and test your application.
11. Write Deposit and Withdraw methods in the SavingAccount class.	<ul style="list-style-type: none"> a. In the SavingAccount class, write a method named Deposit that adds money to the account. Adjust the interest rate accordingly. b. In the SavingAccount class write a method named Withdraw that removes money from the account. Do not permit the balance to fall below zero, and return false if the withdrawal amount is greater than the balance. Adjust the interest rate accordingly.
12. Call the Deposit method from the Form1 class.	<ul style="list-style-type: none"> a. In the deposit_Click method in the Form1 class, write code to deposit 400 into the checking account, and display the new balance. b. Compile, run and test your application.
13. Save your application, and then quit Visual Studio .NET.	<ul style="list-style-type: none"> a. Save your application. b. Quit Visual Studio .NET.