Common C# OOP features

Lesson Objectives

- Understand about params keyword
- Named and Optional Arguments
- Extension Method
- Lambda Expressions
- Coding convention in C#

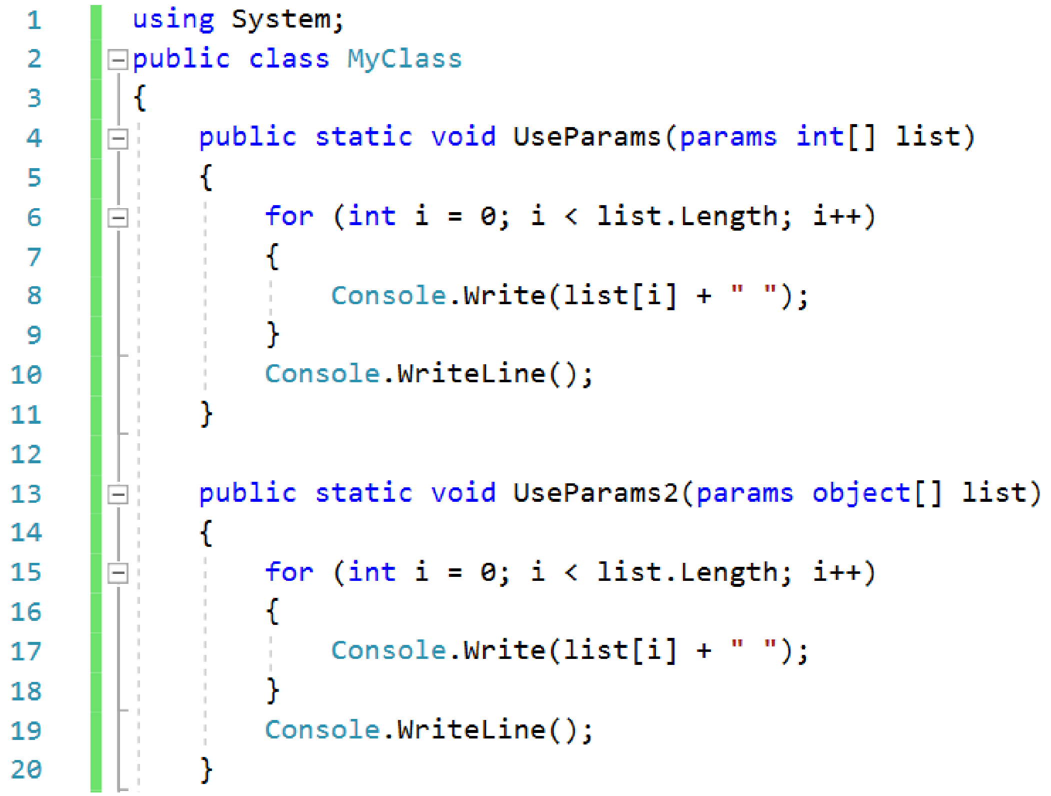
Section 1

PARAMS KEYWORD

Introduction

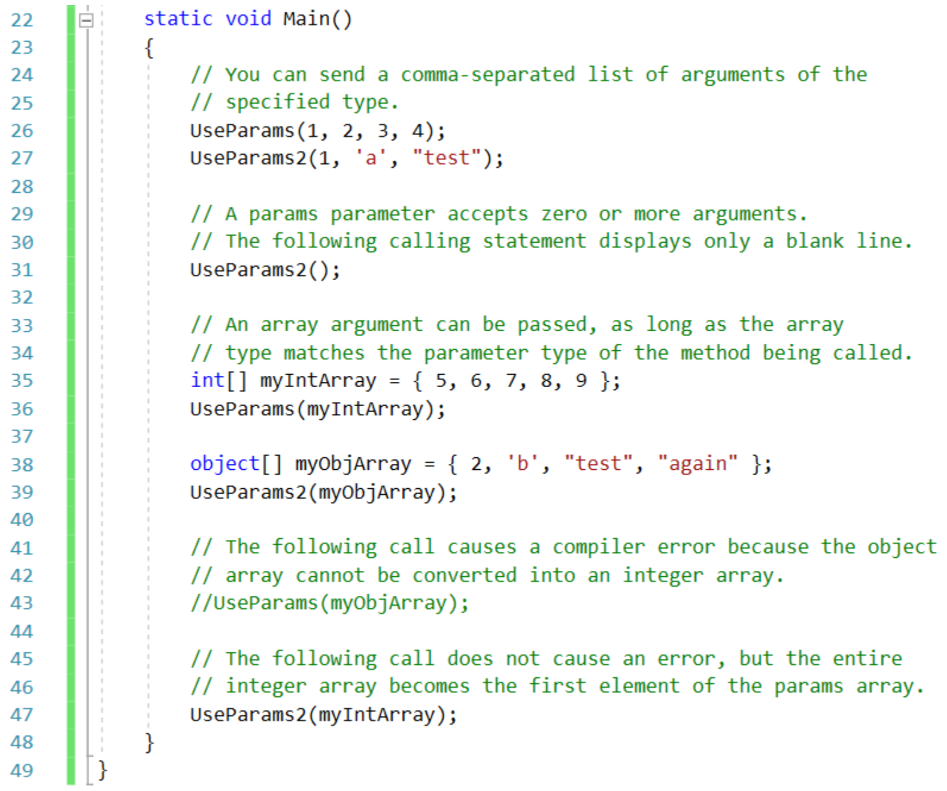
- By using the params keyword, you can specify a method parameter that takes a variable number of arguments. The parameter type must be a single-dimensional array.
- No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration.
- When you call a method with a params parameter, you can pass in:
  - ✓ A comma-separated list of arguments of the type of the array elements.
  - ✓ An array of arguments of the specified type.
  - ✓ No arguments. If you send no arguments, the length of the params list is zero.

Example 1 - 3



```
using System;
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }
```

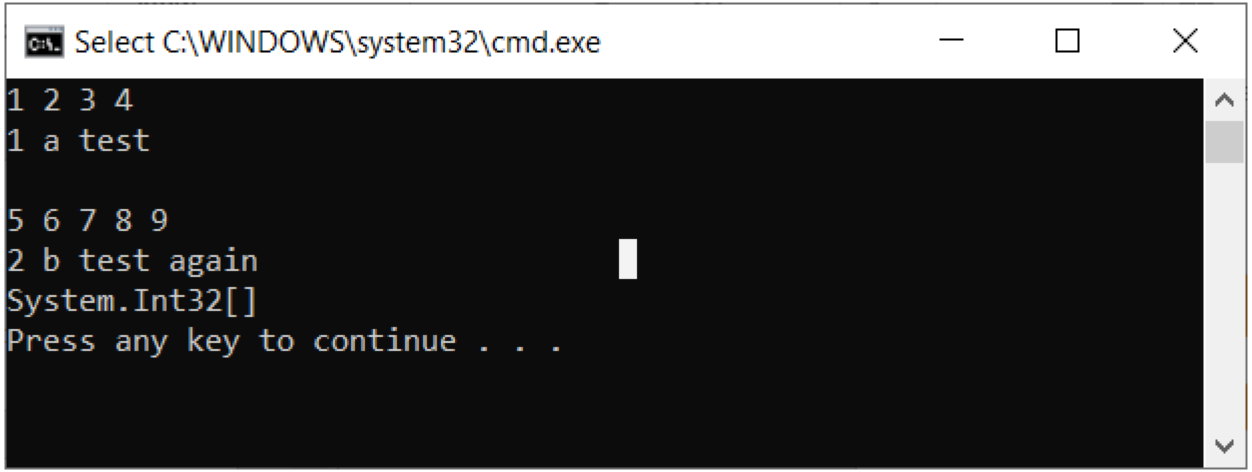
Example 2 - 3



```
    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
```

Example 3 - 3

- Output:



```
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
Press any key to continue . . .
```

Section 2

NAMED AND OPTIONAL ARGUMENTS

Introduction

- C# 4 introduces named and optional arguments
- *Named arguments* enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than with the parameter's position in the parameter list.
- *Optional arguments* enable you to omit arguments for some parameters.
- Both techniques can be used with methods, indexers, constructors, and delegates.

Named Arguments

- Named arguments free you from the need to remember or to look up the order of parameters in the parameter lists of called methods.
- The parameter for each argument can be specified by parameter name.
- For example, a function that prints order details (such as, seller name, order number & product name) can be called in the standard way by sending arguments by position, in the order defined by the function.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

- If you do not remember the order of the parameters but know their names, you can send the arguments in any order.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");

PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

- Named arguments also improve the readability of your code by identifying what each argument represents.

Optional Arguments

- The definition of a method, constructor, indexer, or delegate can specify that its parameters are required or that they are optional.
- Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.
- Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used.

```
public void ExampleMethod(int required, string optionalstr = "default
string", int optionalint = 10)
```

Named and Optional Arguments - Summary

- Named Arguments
- Optional Arguments

Section 3

EXTENSION METHOD

Extension Methods 1-7

- Extension methods allow you to extend an existing type with new functionality without directly modifying those types.
- Extension methods are static methods that have to be declared in a static class.
- You can declare an extension method by specifying the first parameter with the `this` keyword.
- The first parameter in this method identifies the type of objects in which the method can be called.
- The object that you use to invoke the method is automatically passed as the first parameter.

Syntax

```
static return-type MethodName (this type obj, param-list)
```

where:

- `return-type`: the data type of the return value
- `MethodName`: the extension method name
- `type`: the data type of the object
- `param-list`: the list of parameters (optional)

Extension Methods 2-7

- The following code creates an extension method for a string and converts the first character of the string to lowercase:

Snippet

```
using System;

/// <summary>
/// Class ExtensionExample defines the extension method
/// </summary>

static class ExtensionExample
{
        // Extension Method to convert the first character to
        //lowercase
         public static string FirstLetterLower(this string result)
         {
                    if (result.Length > 0){
                          char[] s = result.ToCharArray();
                          s[0] = char.ToLower(s[0]);
                          return new string(s);
                    }
              return result;
              }
}
```

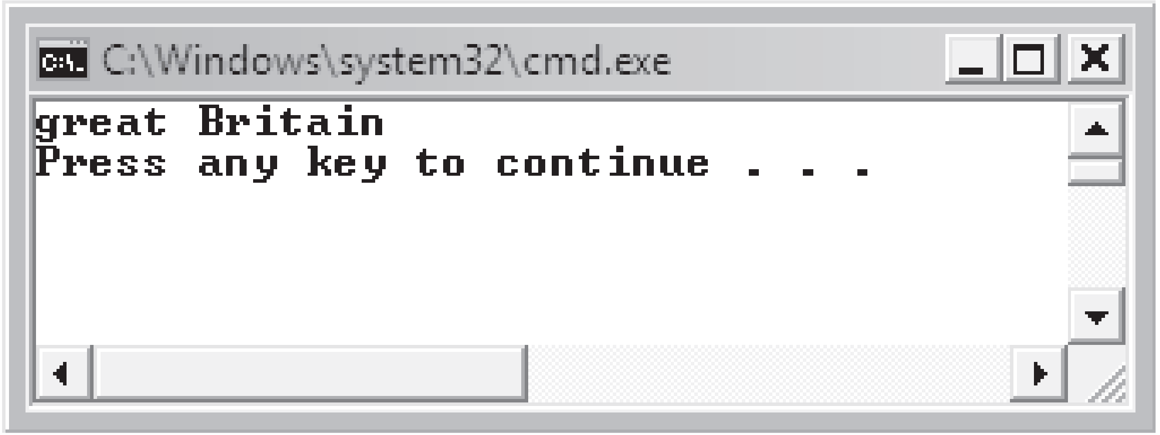
Extension Methods 3-7

```
class Program
{
    public static void Main(string[] args)
    {
      string country = "Great Britain";
      // Calling the extension method
      Console.WriteLine(country.FirstLetterLower());
    }
}
```

- In the code:
  - ✓ An extension method named **`FirstLetterLower`** is defined with one parameter that is preceded with `this` keyword.
  - ✓ This method converts the first letter of any sentence or word to lowercase.
  - ✓ Note that the extension method is invoked by using the object, **`country`**.
  - ✓ The value ‘Great Britain’ is automatically passed to the parameter result.

Extension Methods 4-7

- The following figure depicts the output:



```
great Britain
Press any key to continue . . .
```

- The advantages of extension methods are as follows:
  - ✓ You can extend the functionality of the existing type without modification. This will avoid the problems of breaking source code in existing applications.
  - ✓ You can add additional methods to standard interfaces without physically altering the existing class libraries.

Extension Methods 5-7

- The following code is an example for an extension method that removes all the duplicate values from a generic collection and displays the result.
- This program extends the generic `List` class with added functionality.

Snippet

```
using System;
using System.Collections.Generic;
/// <summary>
/// Class ExtensionExample defines the extension method
/// </summary>
static class ExtensionExample
{
        // Extension method that accepts and returns a collection.
        public static List<T> RemoveDuplicate<T>(this List<T> allCities)
        {
                List<T> finalCities = new List<T>();
                foreach (var eachCity in allCities)
                if (!finalCities.Contains(eachCity))
                finalCities.Add(eachCity);
                return finalCities;
        }
}
```

Extension Methods 6-7

```
class Program
{
        public static void Main(string[] args)
        {
                List<string> cities = new List<string>();
                cities.Add("Seoul");
                cities.Add("Beijing");
                cities.Add("Berlin");
                cities.Add("Istanbul");
                cities.Add("Seoul");
                cities.Add("Istanbul");
                cities.Add("Paris");
                // Invoke the Extension method, RemoveDuplicate().
                List<string> result = cities.RemoveDuplicate();
                foreach (string city in result)
                Console.WriteLine(city);
        }
}
```

- In the code:
  - ✓ The extension method **`RemoveDuplicate()`** is declared and returns a generic `List` when invoked.
  - ✓ The method accepts a generic `List<T>` as the first argument:

```
public static List<T> RemoveDuplicate<T>(this List<T> allCities)
```

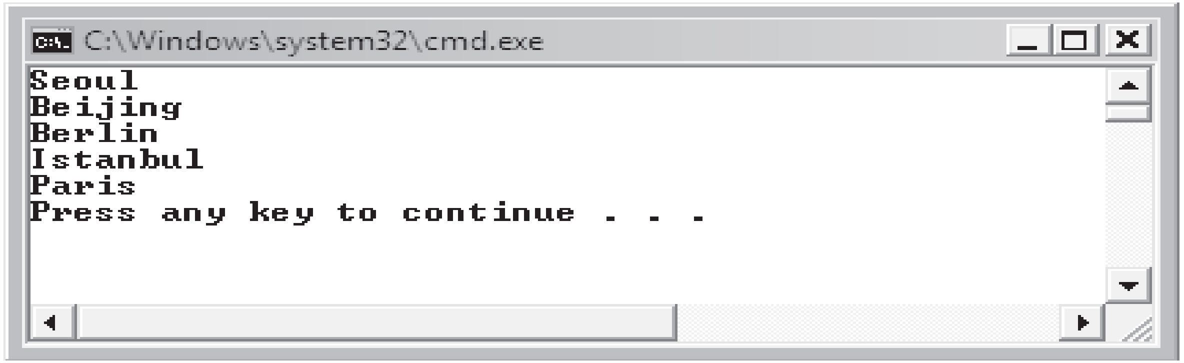
Extension Methods 7-7

- ✓ The following lines of code iterate through each value in the collection, remove the duplicate values, and store the unique values in the `List`, **`finalCities`**:

```
foreach (var eachCity in allCities)
if (!finalCities.Contains(eachCity))
finalCities.Add(eachCity);
```

- ▪ The following figure displays the output:

Output



C:\Windows\system32\cmd.exe

```
Seoul
Beijing
Berlin
Istanbul
Paris
Press any key to continue . . .
```

Extension Methods - Summary

- Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
- The most common extension methods are the LINQ standard query operators that add query functionality to the existing System.Collections.Ienumerable and System.Collections.Generic.IEnumerable<T> types.

Section 4

LAMBDA EXPRESSIONS

Introduction

- A *lambda expression* is an expression of any of the following two forms:
  - ✓ Expression lambda that has an expression as its body:

```
(input-parameters) => expression
```

  - ✓ Statement lambda that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

Expression Lambdas 1-3

- An expression lambda is a lambda with an expression on the right side.

Syntax

```
(input_parameters) => expression
```

where,

`input_parameters`: one or more input parameters, each separated by a comma

`expression`: the expression to be evaluated

- The input parameters may be implicitly typed or explicitly typed.
- When there are two or more input parameters on the left side of the lambda operator, they must be enclosed within parentheses. However, if you have only one input parameter and its type is implicitly known, then the parentheses can be omitted. For example,

```
(str, str1) => str == str1
```

- It means **`str`** and **`str1`** go into the comparison expression which compares `str` with `str1`. In simple terms, it means that the parameters **`str`** and **`str1`** will be passed to the expression **`str`** == **`str1`**.
- Here, it is not clear what are the types of **`str`** and **`str1`**.
- Hence, it is best to explicitly mention their data types:

```
(string str, string str1)=> str==str1
```

Expression Lambdas 2-3

- To use a lambda expression:
  - ✓ Declare a delegate type which is compatible with the lambda expression.
  - ✓ Then, create an instance of the delegate and assign the lambda expression to it. After this, you will invoke the delegate instance with parameters, if any.
  - ✓ This will result in the lambda expression being executed. The value of the expression will be the result returned by the lambda.
- The following code demonstrates expression lambdas:

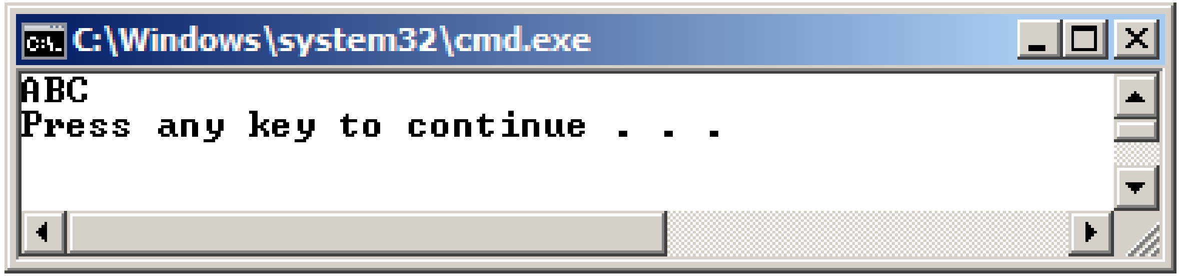
Snippet

```
/// <summary>
/// Class ConvertString converts a given string to uppercase
/// </summary>
public class ConvertString{
        delegate string MakeUpper(string s);
        public static void Main() {
                // Assign a lambda expression to the delegate instance
                MakeUpper con = word => word.ToUpper();
                // Invoke the delegate in Console.WriteLine with a string
                // parameter
                Console.WriteLine(con("abc"));
        }
}
```

Expression Lambdas 3-3

- In the code:
  - ✓ A delegate named **`MakeUpper`** is created and instantiated. At the time of instantiation, a lambda expression, `word => word.ToUpper()` is assigned to the delegate instance.
  - ✓ The meaning of this lambda expression is that, given an input, **`word`**, call the `ToUpper()` method on it.
  - ✓ `ToUpper()` is a built-in method of `String` class and converts a given string into uppercase.
- The following figure displays the output of using expression lambdas:

Output



```
ABC
Press any key to continue . . .
```

Statement Lambdas

- A statement lambda is a lambda with one or more statements. It can include loops, `if` statements, and so forth.

Syntax

```
(input_parameters) => {statement;}
```

- where,
  - ✓ **`input_parameters`**: one or more input parameters, each separated by a comma
  - ✓ **`statement:`** a statement body containing one or more statements
  - ✓ Optionally, you can specify a return statement to get the result of a lambda.
  - ✓ `(string str, string str1)=> { return (str==str1);}`
- The following code demonstrates a statement lambda expression:

Snippet

```
/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
public class WordLength{
// Declare a delegate that has no return value but accepts a string
delegate void GetLength(string s);
public static void Main() {
// Here, the body of the lambda comprises two entire statements
        GetLength len = name => { int n =
        name.Length;Console.WriteLine(n.ToString()); };
        // Invoke the delegate with a string
        len("Mississippi");
        }
}
```

Lambdas with Standard Query Operators 1-2

- Lambda expressions can also be used with standard query operators.
- The following table lists the standard query operators:

Operator	Description
`Sum`	Calculates sum of the elements in the expression
`Count`	Counts the number of elements in the expression
`OrderBy`	Sorts the elements in the expression
`Contains`	Determines if a given value is present in the expression

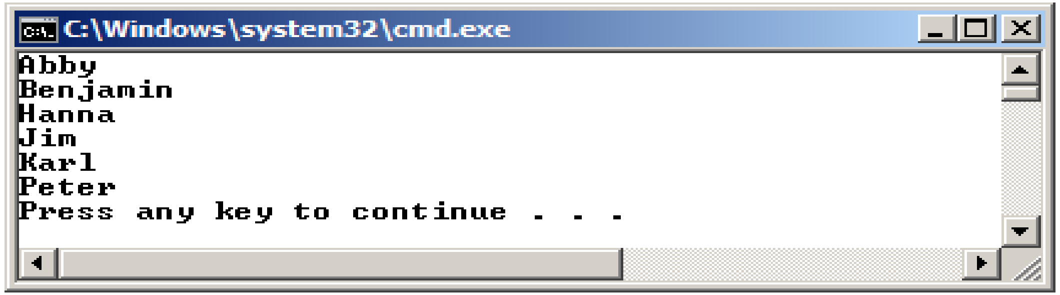
- The following shows how to use the `OrderBy` operator with the lambda operator to sort a list of names:

Snippet

```
/// <summary>
/// Class NameSort sorts a list of names
/// </summary>
public class NameSort{
public static void Main() {
        // Declare and initialize an array of strings
        string [ ] names = {"Hanna", "Jim", "Peter", "Karl", "Abby",
        "Benjamin"};
        foreach (string n in names.OrderBy(name => name)) {
                  Console.WriteLine(n);
        }
    }
}
```

Lambdas with Standard Query Operators 2-2

- The following figure displays the output of the `OrderBy` operator example:



```
Abby
Benjamin
Hanna
Jim
Karl
Peter
Press any key to continue . . .
```

Lambda Expression - Summary

- Use the lambda declaration operator => to separate the lambda's parameter list from its body. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Section 5

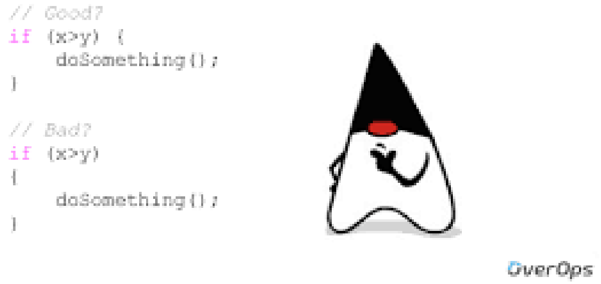
CODING CONVENTION

Introduction

- In computer programming, a naming convention is a set of rules for choosing the character sequence to be used for identifiers which denote variables, types, functions, and other entities in source code and documentation. ()
- Coding conventions serve the following purposes:
  - ✓ They create a consistent look to the code, so that readers can focus on content, not layout.
  - ✓ They enable readers to understand the code more quickly by making assumptions based on previous experience.
  - ✓ They facilitate copying, changing, and maintaining the code.
  - ✓ They demonstrate C# best practices.

Benefits

- Code conventions are important to programmers for a number of reasons
  - ✓ 80% lifetime software cost is for maintenance
  - ✓ People maintain the software may be changed
  - ✓ Following coding convention strictly helps:
    - Improve the readability of the software
    - Allowing engineers to understand new code more quickly and thoroughly



```
// Good?
if (x>y) {
    doSomething();
}

// Bad?
if (x>y)
{
    doSomething();
}
```

Layout Conventions

- Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:
  - ✓ Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces).
  - ✓ Write only one statement per line.
  - ✓ Write only one declaration per line.
  - ✓ If continuation lines are not indented automatically, indent them one tab stop (four spaces).
  - ✓ Add at least one blank line between method definitions and property definitions.
  - ✓ Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Commenting Conventions

- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

```
// The following declaration creates a query. It does not run
// the query.
```

- Do not create formatted blocks of asterisks around comments.

Naming Conventions 1 - 13

- Use **PascalCasing** for class names and method names.

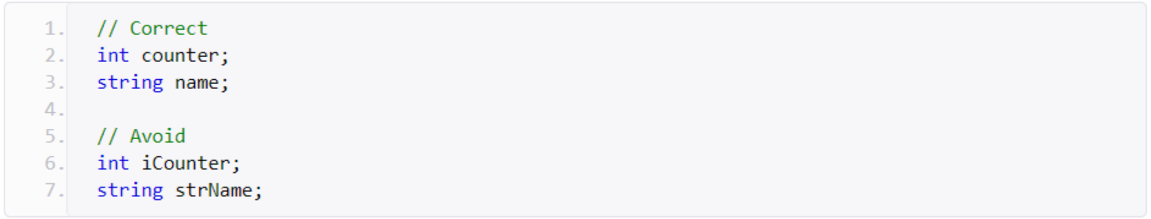
```
public class ClientActivity
{
    public void ClearStatistics()
    {
        //...
    }
    public void CalculateStatistics()
    {
        //...
    }
}
```

- Use **camelCasing** for method arguments and local variables.

```
public class UserLog
{
    public void Add(LogEvent logEvent)
    {
        int itemCount = logEvent.Items.Count;
        // ...
    }
}
```

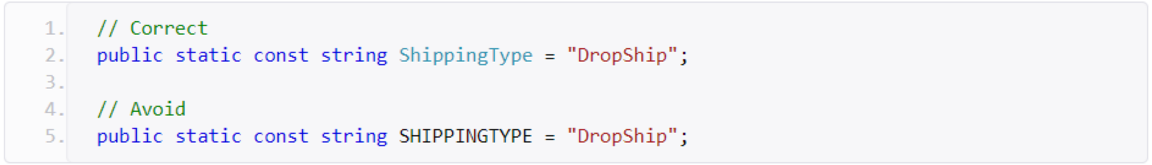
Naming Conventions 2 - 13

- **Do not** use **Hungarian** notation or any other type identification in identifiers



```
// Correct
int counter;
string name;

// Avoid
int iCounter;
string strName;
```

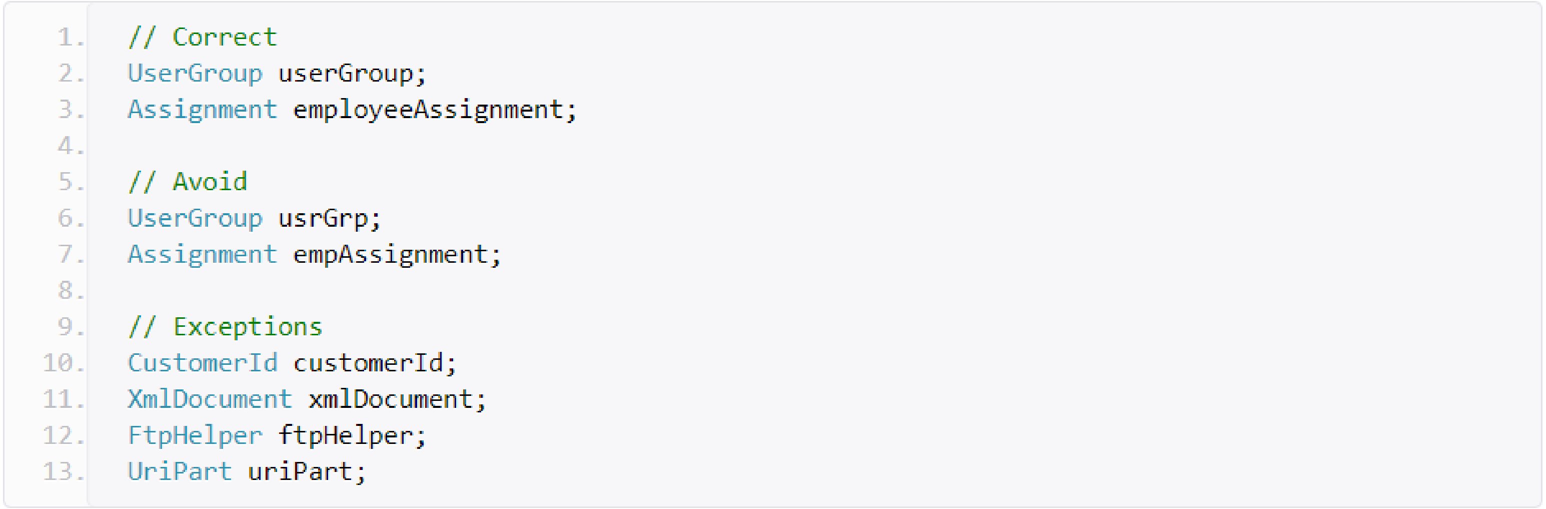
- **Do not** use **Screaming Caps** for constants or readonly variables



```
// Correct
public static const string ShippingType = "DropShip";

// Avoid
public static const string SHIPPINGTYPE = "DropShip";
```

Naming Conventions 3 - 13

- **Avoid** using **Abbreviations**. Exceptions: abbreviations commonly used as names, such as **Id, Xml, Ftp, Uri**



```
// Correct
UserGroup userGroup;
Assignment employeeAssignment;

// Avoid
UserGroup usrGrp;
Assignment empAssignment;

// Exceptions
CustomerId customerId;
XmlDocument xmlDocument;
FtpHelper ftpHelper;
UriPart uriPart;
```

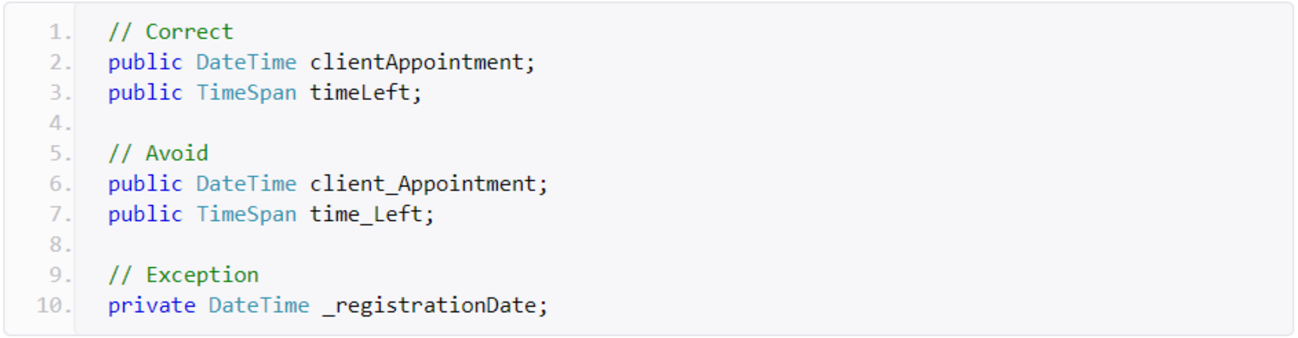
Naming Conventions 4 - 13

- Use **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

```
HtmlHelper htmlHelper;
FtpTransfer ftpTransfer;
UIControl uiControl;
```

Naming Conventions 5 - 13

- **Do not** use **Underscores** in identifiers. Exception: you can prefix private static variables with an underscore.



```
// Correct
public DateTime clientAppointment;
public TimeSpan timeLeft;

// Avoid
public DateTime client_Appointment;
public TimeSpan time_Left;

// Exception
private DateTime _registrationDate;
```

Naming Conventions 6 - 13

- Use **predefined type names** instead of system type names like Int16, Single, UInt64, etc

```csharp
// Correct
string firstName;
int lastIndex;
bool isSaved;

// Avoid
String firstName;
Int32 lastIndex;
Boolean isSaved;
```

- Use implicit type **var** for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

```csharp
var stream = File.Create(path);
var customers = new Dictionary();

// Exceptions
int index = 100;
string timeSheet;
bool isCompleted;
```

Naming Conventions 7 - 13

- Use noun or noun phrases to name a class.

```
public class Employee
{
}
public class BusinessLocation
{
}
public class DocumentCollection
{
}
```

- Do prefix interfaces with the letter **I**. Interface names are noun (phrases) or adjectives.

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

Naming Conventions 8 - 13

- Do name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

```csharp
// Located in Task.cs
public partial class Task
{
    //...
}
```

```csharp
// Located in Task.generated.cs
public partial class Task
{
    //...
}
```

Naming Conventions 9 - 13

- Organize namespaces with a clearly defined structure

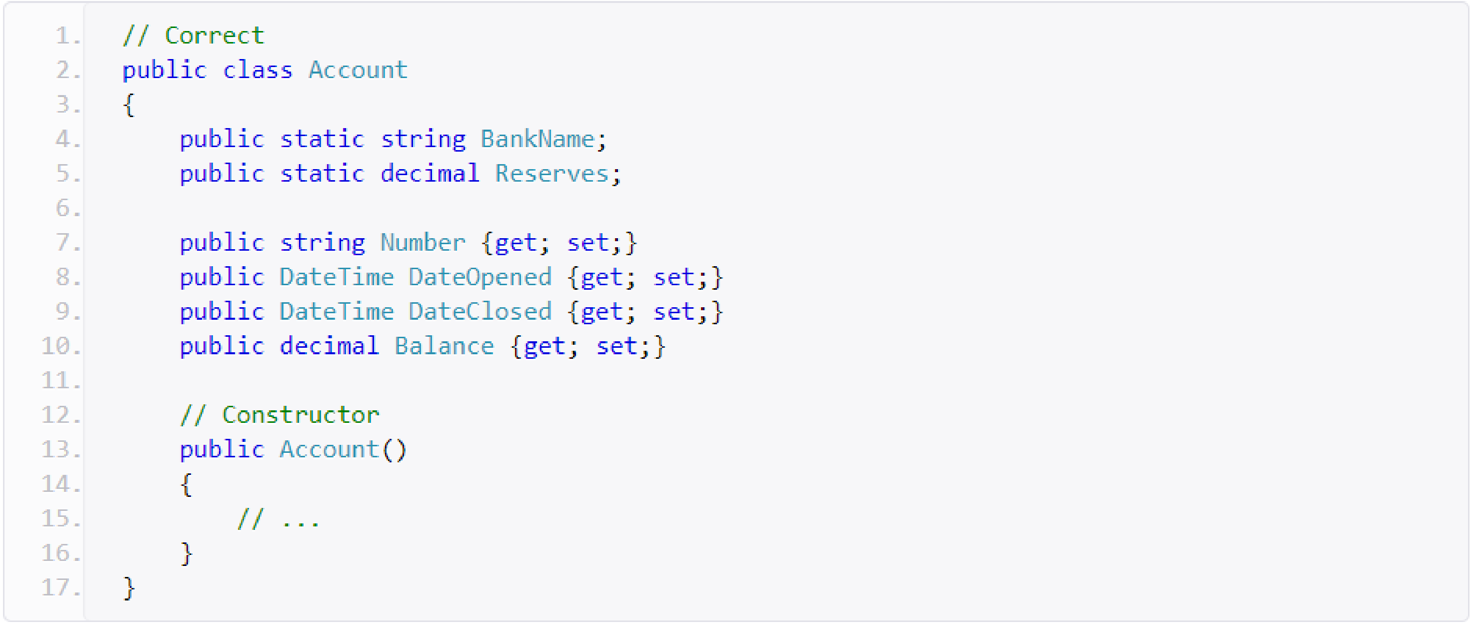
```
// Examples
namespace Company.Product.Module.SubModule
namespace Product.Module.Component
namespace Product.Layer.Module.Group
```

- Vertically align curly brackets.

```
// Correct
class Program
{
    static void Main(string[] args)
    {
    }
}
```

Naming Conventions 10 - 13

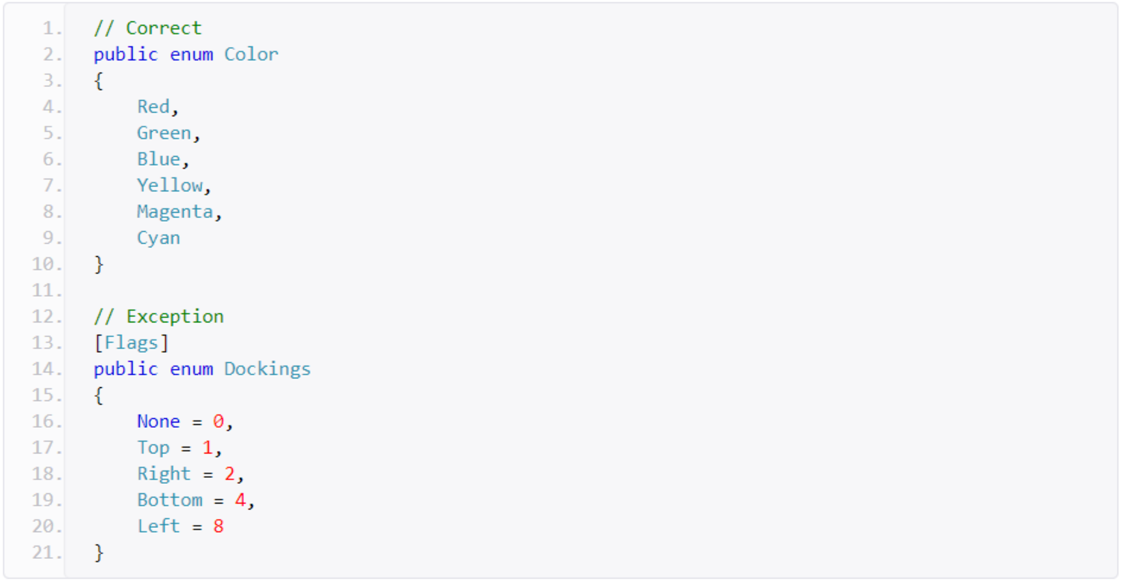
- Declare all member variables at the top of a class, with static variables at the very top.



```
// Correct
public class Account
{
    public static string BankName;
    public static decimal Reserves;

    public string Number {get; set;}
    public DateTime DateOpened {get; set;}
    public DateTime DateClosed {get; set;}
    public decimal Balance {get; set;}

    // Constructor
    public Account()
    {
        // ...
    }
}
```

Naming Conventions 11 - 13

- Use singular names for enums. Exception: bit field enums.



```
// Correct
public enum Color
{
    Red,
    Green,
    Blue,
    Yellow,
    Magenta,
    Cyan
}

// Exception
[Flags]
public enum Dockings
{
    None = 0,
    Top = 1,
    Right = 2,
    Bottom = 4,
    Left = 8
}
```

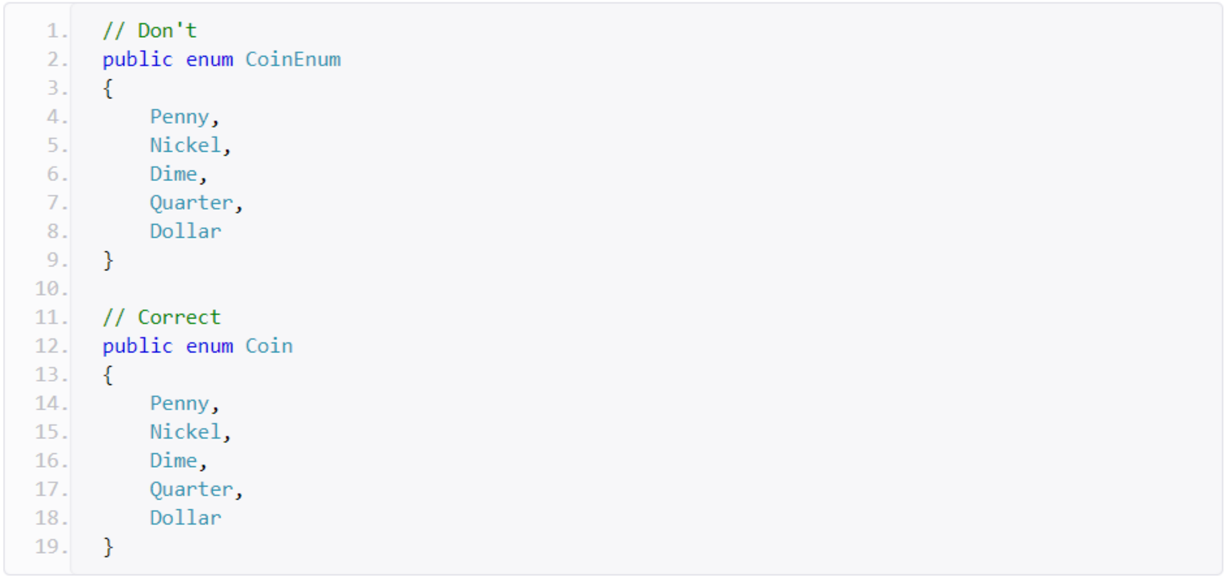
Naming Conventions 12 - 13

- **Do not** explicitly specify a type of an enum or values of enums (except bit fields)

```
// Don't
public enum Direction : long
{
    North = 1,
    East = 2,
    South = 3,
    West = 4
}

// Correct
public enum Direction
{
    North,
    East,
    South,
    West
}
```

Naming Conventions 13 - 13

- **Do not** suffix enum names with Enum



```
// Don't
public enum CoinEnum
{
    Penny,
    Nickel,
    Dime,
    Quarter,
    Dollar
}

// Correct
public enum Coin
{
    Penny,
    Nickel,
    Dime,
    Quarter,
    Dollar
}
```

Language Guidelines- String Data Type

- Use string interpolation to concatenate short strings, as shown in the following code.

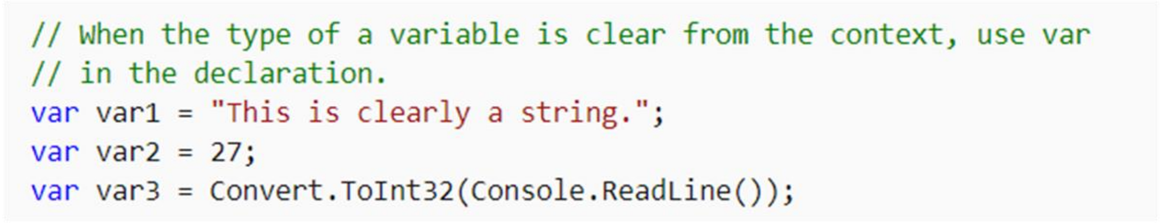
```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- To append strings in loops, especially when you are working with large amounts of text, use a StringBuilder object.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

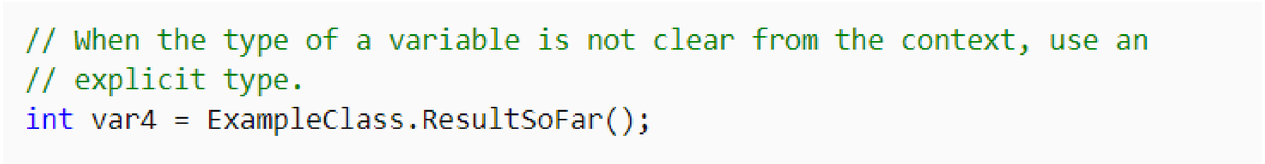
Language Guidelines- Implicitly Typed Local Variables

- Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.



```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- Do not use var when the type is not apparent from the right side of the assignment.



```
// When the type of a variable is not clear from the context, use an
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- Avoid the use of var in place of dynamic.
- Use implicit typing to determine the type of the loop variable in for loops.
- Do not use implicit typing to determine the type of the loop variable in foreach loops.

Lesson Summary

- How to use params keyword.
- Named and Optional Arguments
- Extension methods allow you to extend different types with additional static methods.
- Lambda Expressions
- Coding convention in C#

Thank you