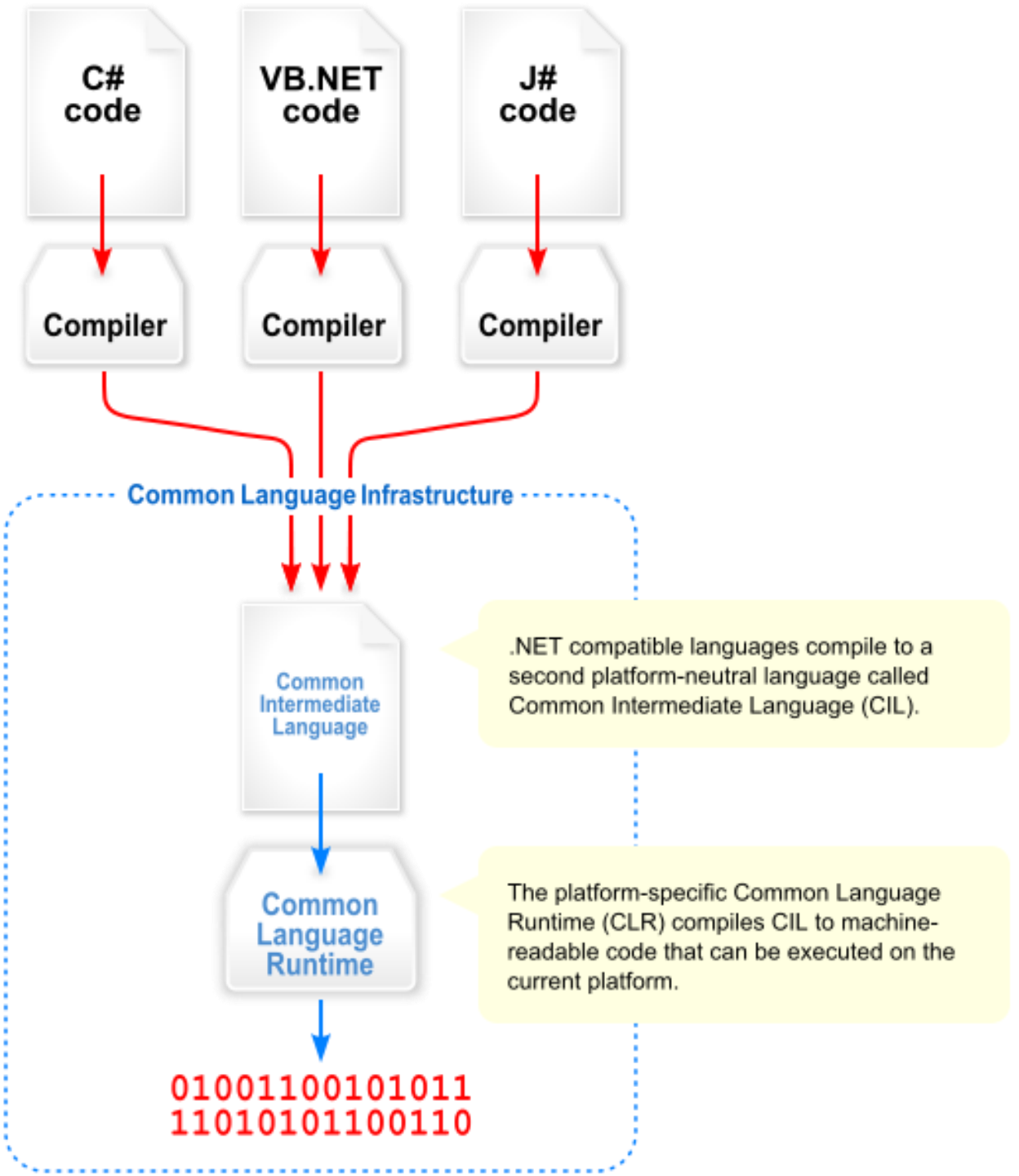
IAP C# and .NET 2011
Lecture 1: Basic Syntax

Geza Kovacs

CLI Languages

.NET Platform



Why target the .NET Platform?

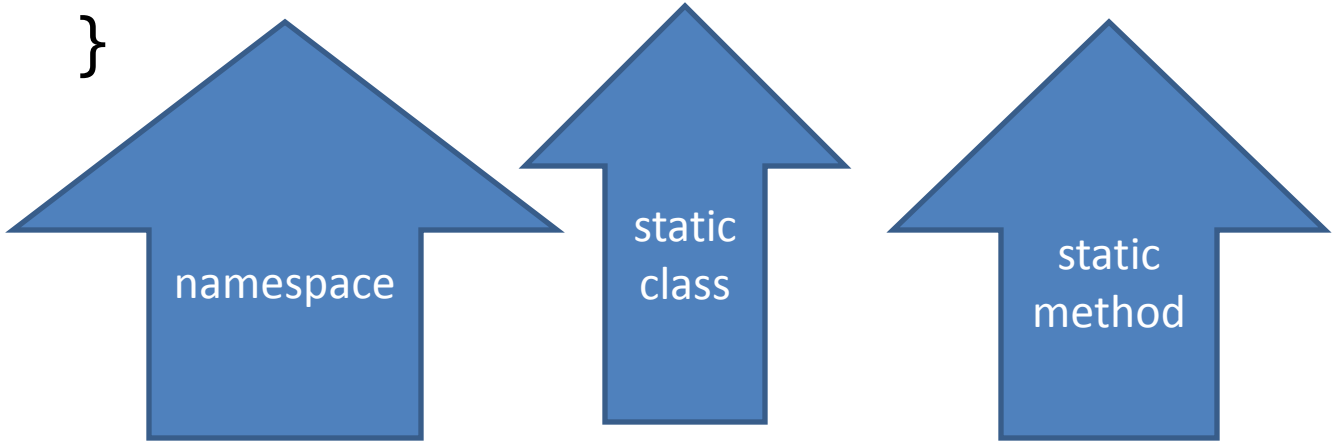
- Applications run on Windows, Silverlight, Zune, Windows Phone 7, Xbox 360, and (via Mono) on Linux, Mac OS X, iPhone/iPad (MonoTouch), and Android (MonoDroid)
- Applications and libraries can be developed in a number of languages
  - **CLI Languages:** C#, C++/CLI, Visual Basic, F#, etc; http://en.wikipedia.org/wiki/CLI_Languages

Why develop in C#?

- Most commonly used CLI language
- Syntax will be familiar to Java and C++ programmers
  - But provides many unique syntactic features of its own: Lambdas, Language Integrated Queries, etc
- Visual Studio provides code completion, refactoring, etc
  - Students can get Visual Studio 2010 Pro for free from Microsoft's DreamSpark site, http://www.dreamspark.com/
  - Or, if on Linux or Mac OS X, use MonoDevelop

Hello World in C#

```
class MyMainClass
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Hello world");
    }
}
```



Hello World in C#

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello world");
    }
}
```

With a using statement, no longer have to type out namespace when referring to something in the namespace

Declaring Variables in C#

- C# is a statically typed language – a variable's type is determined when it's declared
- Can declare types like **int x = 5;**
- Can also make use of type inference: **var x = 5;**
- If you absolutely must have dynamic typing, can use **dynamic x = 5;**

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        int x = 3; // ok
        Console.WriteLine(x);
    }
}
```

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        var x = 3; // ok
        Console.WriteLine(x);
    }
}
```

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        int x; // ok
        x = 3;
        Console.WriteLine(x);
    }
}
```

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        var x; // error
        x = 3;
        Console.WriteLine(x);
    }
}
```

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        var x = 3;
        x = "someString"; // error
        Console.WriteLine(x);
    }
}
```

```
using System;

class MyMainClass
{
    static void Main(string[] args)
    {
        dynamic x = 3;
        x = "someString"; // ok
        Console.WriteLine(x);
    }
}
```

Arrays

- 3 types: 1-dimensional, Multidimensional, and Jagged
  - Multidimensional: a generalized matrix (better performance-wise)
  - Jagged: An Array of Arrays

```
int[] oneDim = new int[] {1, 2, 3}; // 3-element array
int[,] twoDim = new int[, ] {{1,2,3},{4,5,6}}; // 2x3
matrix
int[, ,] threeDim = new int[3, 5, 7]; // 3x5x7 cube
int[][] jagged = new int[3][]; // jagged array
int[,][, ,][] complicatedArray = new int[3, 5][, ,][];
```

```
using System;
class MyMainClass
{
    static void Main(string[] args)
    {
        int[] oneDim = new int[3]; // 3-element array
        for (int i = 0; i < oneDim.Length; ++i)
        {
            oneDim[i] = i;
        }
        foreach (var x in oneDim)
        {
            Console.WriteLine(x);
        }
    }
}
```

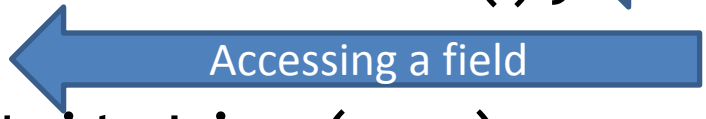
```
using System;
class MyMainClass
{
    static void Main(string[] args)
    {
        int[,] twoDim = new int[3, 5]; // 3x5 matrix
        Console.WriteLine(twoDim.GetLength(0)); // 3
        Console.WriteLine(twoDim.GetLength(1)); // 5
        for (int i = 0; i < twoDim.GetLength(0); ++i)
        {
            for (int j = 0; j < twoDim.GetLength(1); ++j)
            {
                twoDim[i, j] = i * twoDim.GetLength(1) + j;
            }
        }
        foreach (var x in twoDim)
        {
            Console.WriteLine(x); // 0 to 14, increasing
        }
    }
}
```

```
using System;
class MyMainClass
{
    static void Main(string[] args)
    {
        int[][] jagged = new int[3][];
        int counter = 0;
        for (int i = 0; i < jagged.Length; ++i)
        {
            jagged[i] = new int[i+5];
            for (int j = 0; j < jagged[i].Length; ++j)
            {
                jagged[i][j] = counter++;
            }
        }
        foreach (int[] x in jagged)
        {
            foreach (int y in x)
            {
                Console.WriteLine(y); // 0 to 17, increasing
            }
        }
    }
}
```

Classes

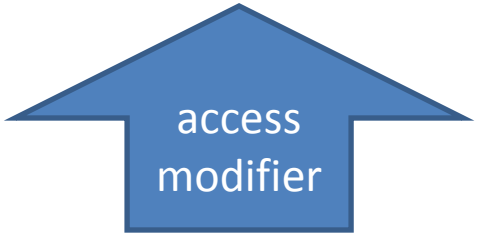
- In C#, all code and fields must be in either a class or struct
- Create an instance of a class with **new**
- Access fields of a class with variable.fieldName

```
using System;

class SomeClass
{
    public int x;
}

class MyMainClass
{
    static void Main(string[] args)
    {
        var s = new SomeClass();
        s.x = 3;
        Console.WriteLine(s.x);
    }
}
```



Access Modifiers

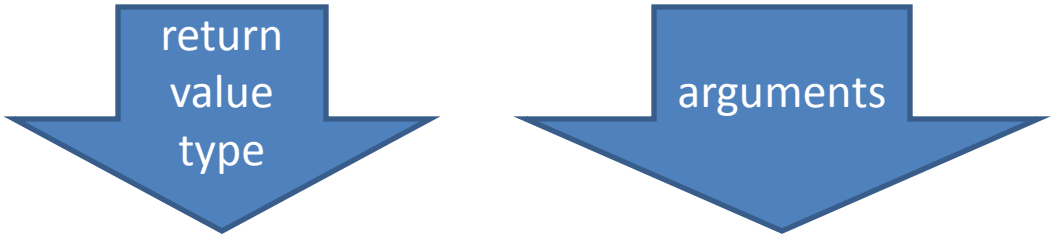
```
class SomeClass
{
    public int x;
}
```



- Access modifiers: can be one of
  - private (default for fields and methods): accessible only within the class
  - public: accessible from anywhere
  - There are also protected, internal, and protected internal

Methods

- Take 0 or more arguments, do some computation, return 0 or 1 values
- Can be instance methods, or static methods
  - Instance methods can access fields of the class instance, static methods cannot




```
static bool IsEven(int a) {
    …
}
```

```
using System;
class MathStuff
{
    public static bool IsEven(int a)
    {
        if (a % 2 == 0) {
            return true;
        } else {
           return false;
        }
    }
}
class MyMainClass
{
   static void Main(string[] args)
   {
       int x = 5;
       Console.WriteLine(MathStuff.IsEven(x));
    }
}
```

static method

static method

Static method invocation: don't need to create instance beforehand

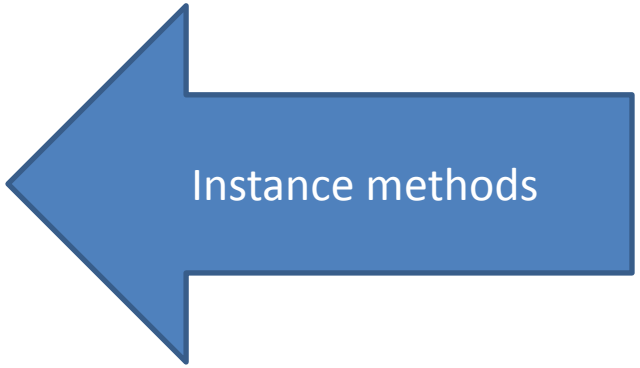
```
using System;
static class MathStuff
{
    public static bool IsEven(int a)
    {
        if (a % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }
}
static class MyMainClass
{
   static void Main(string[] args)
   {
       int x = 5;
       Console.WriteLine(MathStuff.IsEven(x));
    }
}
```

If a class has only static fields and methods, it can be made static

- Static classes cannot be instantiated nor inherited from

Same with this class

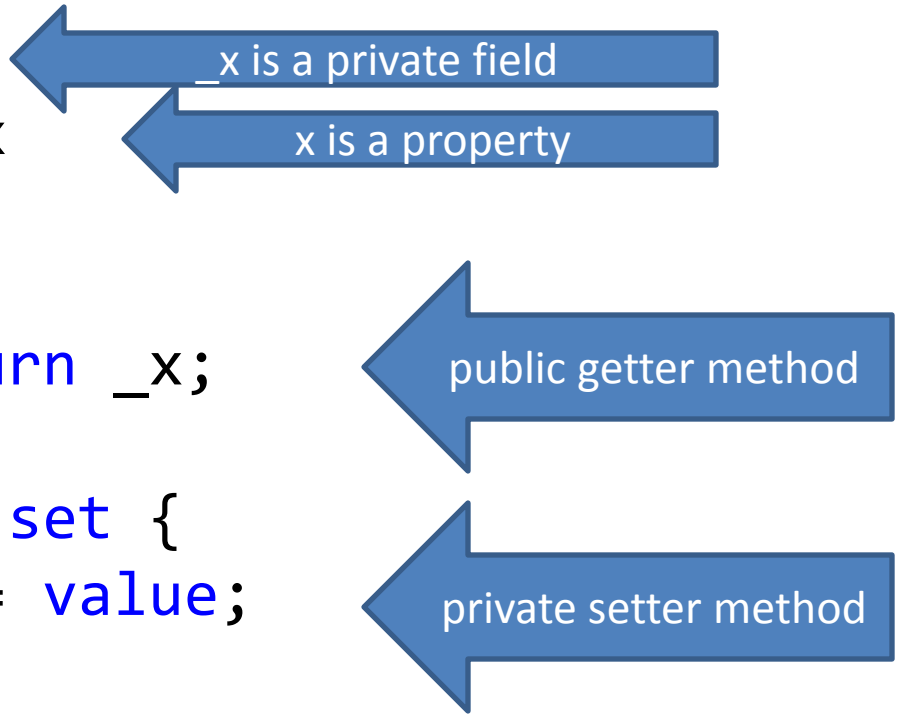
```
using System;

class Counter
{
    int x = 0;
    public int GetX() { return x; }
    public void Increment() { ++x; }
    public void Decrement() { --x; }
}

static class MyMainClass
{
    static void Main(string[] args)
    {
        var c = new Counter();
        c.Increment();
        Console.WriteLine(c.GetX());
    }
}
```



Properties

- Getters and Setter methods are often used in other languages for controlling access to fields
- In C#, we usually use properties for this purpose

```
int _x = 0;
public int x
{
    get {
        return _x;
    }
    private set {
        _x = value;
    }
}
```



Properties

- Getters and Setter methods are often used in other languages for controlling access to fields
- In C#, we usually use properties for this purpose

```
public int x {
    get;
    private set;
}
```

```
using System;

class Counter
{
    public int x { get; private set; }
    public void Increment() { ++x; }
    public void Decrement() { --x; }
}

static class MyMainClass
{
    static void Main(string[] args)
    {
        var c = new Counter();
        c.Increment();
        Console.WriteLine(c.x);
        c.x = 3; // setting c.x is not allowed
    }
}
```

Method Overloading

- So long as method signatures differ (in argument type, or number of arguments), you can define multiple methods with the same name

```
static int add(int a, int b) {…}
static float add(float a, float b) {…}
static double add(double a, double b) {…}
static int add(int a, int b, int c) {…}
```

```
using System;
static class MyMainClass
{
    static int add(int a, int b) {
        return a + b;
    }
    static float add(float a, float b) {
        return a + b;
    }
    static double add(double a, double b) {
        return a + b;
    }
    static int add(int a, int b, int c) {
        return a + b + c;
    }
    static void Main(string[] args)  {
        Console.WriteLine(add(2.2, 3.3)); // 5.5
        Console.WriteLine(add(2.2f, 3.3f)); // 5.5
        Console.WriteLine(add(3, 6)); // 9
        Console.WriteLine(add(3, 6, 7)); // 16
    }
}
```

Variable Number of Arguments

- Suppose you want to implement a **sum()** method which returns the sum of its arguments
  - That is, sum()=0, sum(3)=3, sum(3,6,7)=16, etc
- Use the **params** keyword to do this; the list of arguments is exposed to the method as an array

```
static int sum(params int[] argsList) {…}
```

```
using System;

static class MyMainClass
{
    static int sum(params int[] argsList)
    {
        int total = 0;
        foreach (int x in argsList)
        {
            total += x;
        }
        return total;
    }
    static void Main(string[] args)  {
        Console.WriteLine(sum()); // 0
        Console.WriteLine(sum(3)); // 3
        Console.WriteLine(sum(3, 6, 7)); // 16
    }
}
```

Pass by Reference

- Recall that a return statement can return at most 1 value
- One way to return multiple values is by passing references to the method invocation.
- Accomplish this by marking the marking the method argument as either:
  - **out** if the argument has not yet been initialized (will not be able to read its value in the method, only set it)
  - **ref** if the argument has been initialized (will be able to both read and set the argument's value in the method)

```
using System;
static class MyMainClass
{
    static int divide(int numerator,
                      int denominator,
                      out int remainder) {
        remainder = numerator % denominator;
        return numerator / denominator;
    }

    static void Main(string[] args)
    {
        var num = 14;
        var den = 4;
        int rem;
        var result = divide(num, den, out rem);
        Console.WriteLine("result: " + result);
        Console.WriteLine("rem: " + rem);
    }
}
```

add “out” in signature

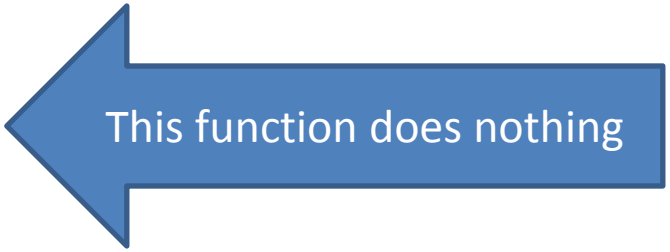
Mark argument as “out” in invocation

```
using System;

static class MyMainClass
{
    static void swap(ref int x, ref int y) {
        var t = x;
        x = y;
        y = t;
    }

    static void Main(string[] args)
    {
        var q = 5;
        var r = 7;
        swap(ref q, ref r);
        Console.WriteLine("q: " + q);
        Console.WriteLine("r: " + r);
    }
}
```

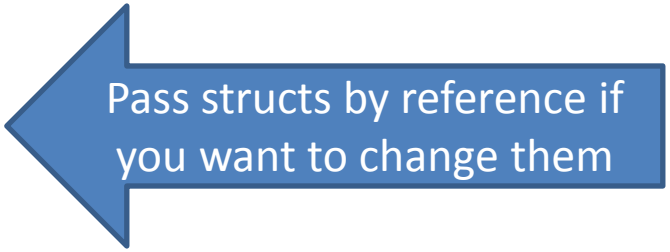
Structs

- Similar in many ways to classes: can have fields, methods, properties, etc
- However, structs (as well as int, double…) are **value types**, whereas classes (as well as arrays, strings…) are **reference types**
  - Value types: allocated on the stack, cannot be assigned null, passes a copy when passed to a function
  - Reference types: allocated on the heap, can be assigned null, passes a reference when passed to a function

```
using System;
struct AStruct { public int x; }
class AClass { public int x; }

static class MyMainClass
{
    static void setValue(AStruct s, int x)
    {
        s.x = x;
    }
    static void setValue(AClass s, int x)
    {
        s.x = x;
    }
    static void Main(string[] args)
    {
        var astruct = new AStruct();
        var aclass = new AClass();
        setValue(astruct, 5);
        setValue(aclass, 5);
        Console.WriteLine(astruct.x); // 0
        Console.WriteLine(aclass.x); // 5
    }
}
```



```
using System;
struct AStruct { public int x; }
class AClass { public int x; }

static class MyMainClass
{
    static void setValue(ref AStruct s, int x)
    {
        s.x = x;
    }
    static void setValue(AClass s, int x)
    {
        s.x = x;
    }
    static void Main(string[] args)
    {
        var astruct = new AStruct();
        var aclass = new AClass();
        setValue(ref astruct, 5);
        setValue(aclass, 5);
        Console.WriteLine(astruct.x); // 5
        Console.WriteLine(aclass.x); // 5
    }
}
```



Constructors

- Called when a class or struct instance is created
- Can accept various arguments

```
class Point
{
    public int x { get; private set; }
    public int y { get; private set; }
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

```
using System;
class Point
{
    public int x { get; private set; }
    public int y { get; private set; }
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point(4, 7);
        Console.WriteLine(p.x); // 4
        Console.WriteLine(p.y); // 7
    }
}
```

```
using System;
class Point
{
    public readonly int x;
    public readonly int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

static class MyMainClass
{
    static void Main(string[] args)
    {
        Point p = new Point(4, 7);
        Console.WriteLine(p.x); // 4
        Console.WriteLine(p.y); // 7
    }
}
```

readonly keyword: Can only be set in the constructor; useful when making immutable objects. Similar to **final** in Java

Operator Overloading

- Motivation: cleaner syntax for operations like addition, multiplication, etc on your custom datatypes
- Ex: If p1 and p2 are Point classes (or structs), can do p1 + p2 to add their coordinates, as opposed to p1.addToPoint(p2)

```
using System;
class Point
{
    public readonly int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public static Point operator + (Point p1, Point p2) {
        return new Point(p1.x + p2.x, p1.y + p2.y);
    }
}

static class MyMainClass
{
    static void Main(string[] args)  {
        Point p1 = new Point(4, 7);
        Point p2 = new Point(3, 9);
        Point p = p1 + p2;
        Console.WriteLine(p.x);
        Console.WriteLine(p.y);
    }
```

Equality operator (==)

For reference types other than string, == returns true if its two operands refer to the same object. For the string type, == compares the values of the strings.

```
using System;
class  Point
{
    public readonly int x, y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}

static class MyMainClass
{
    static void Main(string[] args) {
        Point p1 = new Point(4, 3);
        Point p2 = new Point(4, 3);
        Console.WriteLine(p1 == p2); // False
    }
}
```

```
using System;
class Point
{
    public readonly int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public static bool operator == (Point p1, Point p2) {
        return (p1.x == p2.x && p1.y == p2.y);
    }
    public static bool operator != (Point p1, Point p2) {
        return (p1.x != p2.x || p1.y != p2.y);
    }
}

static class MyMainClass
{
    static void Main(string[] args) {
        Point p1 = new Point(4, 3);
        Point p2 = new Point(4, 3);
        Console.WriteLine(p1 == p2); // True
    }
}
```

If overloading ==, also overload !=

Generics

- Suppose you want to create a Pair class which stores a pair of values, of arbitrary type
- There'll be 2 fields, but what will their types be?
  - Bad solution: dynamic or casting from Object (not type-safe)
  - Good solution: with Generics

```
using System;
class Pair<T, U>
{
    public readonly T Item1;
    public readonly U Item2;
    public Pair(T Item1, U Item2) {
        this.Item1 = Item1; this.Item2 = Item2;
    }
}
static class MyMainClass
{
    static void Main(string[] args) {
        string x = "";
        int y = 5;
        Pair<string, int> z = new Pair<string, int>(x, y);
        string q = z.Item1;
        int r = z.Item2;
    }
}
```

```
using System;
class Pair<T, U>
{
    public readonly T Item1;
    public readonly U Item2;
    public Pair(T Item1, U Item2) {
        this.Item1 = Item1; this.Item2 = Item2;
    }
}
static class MyMainClass
{
    static Pair<T, U> makePair<T, U>(T x, U y) {
        return new Pair<T, U>(x, y);
    }
    static void Main(string[] args) {
        string x = "";
        int y = 5;
        Pair<string, int> z = makePair(x, y);
        string q = z.Item1;
        int r = z.Item2;
    }
}
```

Notes on Generics

- If you need a generic container for pairs (or more) of values, use Tuple<T, U>
- Unlike Java's Generics, .NET Generics can contain both value and reference types

Generic Collections

- Found in the System.Collections.Generic namespace
  - Hash Table (Dictionary), Binary tree (SortedDictionary), LinkedList, etc

```
using System;
using System.Collections.Generic;
using System.Linq;
static class MyMainClass
{
    static void Main(string[] args)
    {
        var list = new LinkedList<int>();
        for (int i = 0; i < 100; ++i)
        {
            list.AddLast(i);
        }
        foreach (int x in list)
        {
            Console.WriteLine(x);
        }
    }
}
```

Extension Methods

- Sometimes your datatypes don't have all the methods you want
  - For example, converting a LinkedList<T> to a T[]
- Usually, you'd write a static method elsewhere (for example, static class Utils), and call Utils.ToArray(linkedlist)
- But wouldn't it be much nicer to just write linkedlist.ToArray()

```
using System;
using System.Collections.Generic;
static class Utils {
    public static T[] ToArray<T>(this LinkedList<T> list) {
        T[] arr = new T[list.Count];
        int i = 0;
        foreach (var x in list) {
            arr[i++] = x;
        }
        return arr;
    }
}
static class MyMainClass {
    static void Main(string[] args) {
        var list = new LinkedList<int>();
        for (int i = 0; i < 100; ++i) {
            list.AddLast(i);
        }
        int[] arr = list.ToArray();
    }
}
```

Extension method for LinkedList

Extension methods can be invoked with the same syntax as a normal method

Higher-Order Functions

- Functions which take functions as arguments
- Example:
  - Map: applies a function to each element in an array
- Func<TInput1, TInput2, TOutput> is a datatype which represents a function; use it for passing functions to higher-order functions

```
using System;
static class MyMainClass
{
    static int square(int x)
    {
        return x * x;
    }
    static int[] map(int[] orig, Func<int, int> fn)
    {
        int[] result = new int[orig.Length];
        for (int i = 0; i < orig.Length; ++i)
        {
            result[i] = fn(orig[i]);
        }
        return result;
    }
    static void Main(string[] args)
    {
        var vals = new int[] { 1, 2, 3, 4, 5 };
        var squared = map(vals, square);
        foreach (var x in squared)
        {
            Console.WriteLine(x);
        }
    }
```

Inheritance

- Motivating Inheritance: if you have 2 classes (for example, Dog and Cat) with the same method (for example, makeNoise), and you have some other method which relies on makeNoise, you will still need separate (but identical) methods for Cat and Dog

```
static void makeLotsOfNoise(Dog x) {
    for (int i = 0; i < 100; ++i) x.makeNoise();
}
static void makeLotsOfNoise(Cat x) {
    for (int i = 0; i < 100; ++i) x.makeNoise();
}
```

```
using System;
class Dog {
    public void makeNoise() { Console.WriteLine("woof"); }
}
class Cat {
    public void makeNoise() { Console.WriteLine("meo"); }
}
static class MyMainClass
{
    static void makeLotsOfNoise(Dog x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void makeLotsOfNoise(Cat x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void Main(string[] args) {
        var dog = new Dog(); var cat = new Cat();
        makeLotsOfNoise(dog);
        makeLotsOfNoise(cat);
    }
}
```

Inheritance

- Solution: create a superclass (ex: Animal) that has that method, and have makeLotsOfNoise accept an Animal as its argument

```
class Dog : Animal {…}
```

Dog inherits Animal (a Dog is an Animal)

```
class Cat : Animal {…}
```

Cat inherits Animal (a Cat is an Animal)

```
static void makeLotsOfNoise(Animal x)
{
    for (int i = 0; i < 100; ++i)
        x.makeNoise();
}
```

```
using System;
class Animal {
    public void makeNoise() { Console.WriteLine("animal"); }
}
class Dog : Animal {
    public void makeNoise() { Console.WriteLine("woof"); }
}
class Cat : Animal {
    public void makeNoise() { Console.WriteLine("meo"); }
}
static class MyMainClass
{
    static void makeLotsOfNoise(Animal x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void Main(string[] args) {
        var dog = new Dog(); var cat = new Cat();
        makeLotsOfNoise(dog);
        makeLotsOfNoise(cat);
    }
}
```

```
using System;
class Animal {
    public void makeNoise() { Console.WriteLine("animal"); }
}
class Dog : Animal {
    public void makeNoise() { Console.WriteLine("woof"); }
}
class Cat : Animal {
    public void makeNoise() { Console.WriteLine("meo"); }
}
static class MyMainClass
{
    static void makeLotsOfNoise(Animal x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void Main(string[] args) {
        var dog = new Dog(); var cat = new Cat();
        makeLotsOfNoise(dog);
        makeLotsOfNoise(cat);
    }
}
```

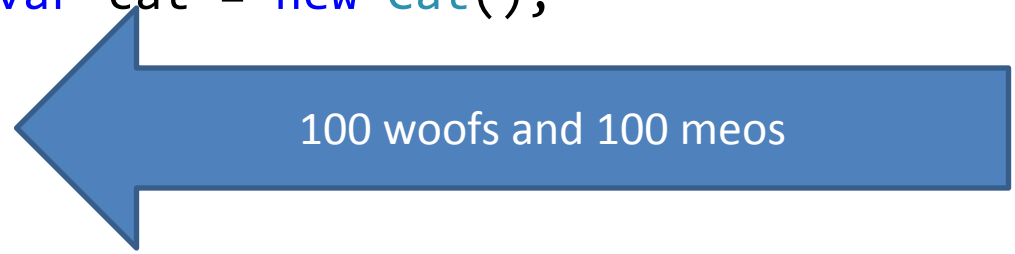
Surprise! “animal” gets printed 200 times, as opposed to 100 woofs and 100 meos

Overriding Methods

- In the previous example, Animal's makeNoise method was not overridden by Cat and Dog
- **virtual** keyword indicates that this method can be overridden by a subclass
- **override** keyword indicates that this method overrides its superclass' implementation

```
public virtual void makeNoise() { Console.WriteLine("animal"); }
public override void makeNoise() { Console.WriteLine("woof"); }
public override void makeNoise() { Console.WriteLine("meo"); }
```

```
using System;
class Animal {
    public virtual void makeNoise() { Console.WriteLine("animal"); }
}
class Dog : Animal {
    public override void makeNoise() { Console.WriteLine("woof"); }
}
class Cat : Animal {
    public override void makeNoise() { Console.WriteLine("meo"); }
}
static class MyMainClass
{
    static void makeLotsOfNoise(Animal x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void Main(string[] args) {
        var dog = new Dog(); var cat = new Cat();
        makeLotsOfNoise(dog);
        makeLotsOfNoise(cat);
    }
}
```



```
using System;
abstract class Animal {
    public abstract void makeNoise();
}
class Dog : Animal {
    public override void makeNoise() { Console.WriteLine("woof"); }
}
class Cat : Animal {
    public override void makeNoise() { Console.WriteLine("meo"); }
}
static class MyMainClass
{
    static void makeLotsOfNoise(Animal x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void Main(string[] args) {
        var dog = new Dog(); var cat = new Cat();
        makeLotsOfNoise(dog);
        makeLotsOfNoise(cat);
    }
}
```

Alternatively, use an abstract class if you don't actually need the implementation

100 woofs and 100 meos

Notes on Subclassing

- A class can inherit from another class, or from a struct
- However, a struct cannot inherit from other structs or classes
- Use the **sealed** keyword to prevent anyone from inheriting from a particular class
- A class can inherit from only one other (abstract or standard) class or struct

Interfaces

- Multiple interfaces can be implemented
- Can include properties and methods
  - However, properties and methods can only be declared, not implemented
- Interface methods are public by default, and don’t need to be explicitly overridden
- In .NET naming convention, interfaces start with the letter “I”

```
using System;
interface IAnimal {
    void makeNoise();
}
class Dog : IAnimal {
    public void makeNoise() { Console.WriteLine("woof"); }
}
class Cat : IAnimal {
    public void makeNoise() { Console.WriteLine("meo"); }
}
static class MyMainClass
{
    static void makeLotsOfNoise(IAnimal x) {
        for (int i = 0; i < 100; ++i) x.makeNoise();
    }
    static void Main(string[] args) {
        var dog = new Dog(); var cat = new Cat();
        makeLotsOfNoise(dog);
        makeLotsOfNoise(cat);
    }
}
```

No need to mark interface methods as virtual

No need to override interface methods

100 woofs and 100 meos